MASTER IN COMPUTER SCIENCE

Utilisation des vues pour l'évolution des applications de bases de données relationnelles

Aerts, Cédric

Award date:
2009

Facultés Universitaires Notre-Dame de la Paix
Faculté d'Informatique

Année Académique 2008-2009

Utilisation des vues pour l'évolution des applications de bases de données relationnelles

Cédric AERTS

Mémoire présenté en vue de l'obtention
du grade de Licencié en Informatique.

Résumé

Au cours de sa vie, une base de données est amenée à être modifiée pour répondre à l'évolution des besoins des utilisateurs. Suite à ces modifications, il faut pouvoir garantir la compatibilité avec le logiciel existant. Ce mémoire a pour objectif d'explorer la solution qui consiste à utiliser le mécanisme des vues relationnelles pour réduire au mieux l'impact des changements de la base de données sur le code applicatif. Il propose un outil de génération automatique de vues en fonction de la nature de la transformation rencontrée.

Mots clés : base de données relationnelle, évolution, vue, transformation.

Abstract

During its lifetime, a database is bound to be revised to match the evolution of its users' needs. Once revised, the backward compatibility of the existing software should be guaranteed. This thesis aims at exploring the potential of relational views to limit the incidence of database changes onto the application code. It presents a tool automatically generating views according to the listed changes.

Keywords : relational database, evolution, view, transformation.

Remerciements

Merci

---

au Professeur Jean-Luc Hainaut, promoteur de ce mémoire, pour m'avoir fait confiance et m'avoir permis de réaliser ce travail dans le domaine des bases de données.

à Anthony Cleve, co-promoteur, qui, malgré un agenda très chargé, a su trouver du temps à me consacrer. Son expertise dans le domaine, ses explications, et ses remarques judicieuses tout au long de la réalisation de ce mémoire m'ont été d'une aide indispensable.

à Anne-Laurence pour sa patience et son soutien au cours de ces trois longues années.

à mes parents, mes soeurs et mon frère pour m'avoir toujours encouragé.

à Radu Cotet qui, en plus de m'avoir permis de concilier travail et étude, a toujours été là pour me (re)motiver.

à mes amis.

à tous les membres du personnel de la Faculté d'Informatique. J'ai eu la chance de faire mes études et mon mémoire dans un environnement universitaire, leurs différents conseils et remarques au détour d'une pause café ont été très enrichissants.

A mes petits coeurs, Léa et Théo...

Table des matières

Première partie

Introduction

Introduction

Motivation

Le monde est en perpétuelle évolution, les systèmes d'information, systèmes qui traitent et exploitent les informations d'une organisation, n'échappent pas à la règle. Les données de tels systèmes sont stockées et gérées via une ou plusieurs bases de données. Au cours de leur vie, ces bases de données sont donc amenées à être modifiées pour répondre à l'évolution des besoins des utilisateurs. Face à ces modifications, l'administrateur de la base de données est confronté à plusieurs difficultés. En effet, ces modifications ont des répercutions aussi bien sur la structure, sur les données, que sur le code applicatif. Il existe quelques méthodes et outils d'aide à la réalisation du processus d'évolution, malheureusement, ils sont si peu nombreux que la personne en charge de cette lourde tâche se retrouve souvent seule avec encore un grand nombre de problèmes à résoudre (migration des données, modification des programmes, estimation du temps de la migration, évaluation des impacts,...).

Nous pouvons distinguer trois aspects lors du processus d'évolution d'une base de données [HH06] :

- **La modification du schéma**
  Certains logiciels, comme DB-Main [dbm], fournissent des outils permettant de construire notamment des schémas conceptuels, de les transformer et de les traduire automatiquement en code exploitable.
- **La migration des données**
  Suite aux modifications de la base de données, il faut adapter les données de manière à les rendre compatibles avec la nouvelle structure. Pour ce faire, il faudra extraire les données, les transformer, et enfin les insérer dans la base de données modifiée. Tout ceci, en prenant soin de conserver leur intégrité et en évitant d'éventuelles pertes d'information.
- **L'adaptation des programmes**
  Il faut aussi adapter le code applicatif. Certaines études montrent en effet que le pourcentage de requêtes devant être réécrites suite à une modification du schéma peut atteindre 70% [CMTZ08]. Par exemple, si une table change de nom, les requêtes qui y faisaient références ne vont plus aboutir et vont retourner un message d'erreur. Ces adaptations peuvent s'avérer très complexes et coûteuses.

Contexte de recherche

Un grand nombre de travaux traitent des différents aspects de l'évolution des bases de données. Elles apportent de bonnes bases théoriques et proposent des démarches méthodologiques, mais malheureusement peu d'entre elles aboutissent à l'implémentation d'un outil concret permettant d'assister l'administrateur dans sa lourde tâche.
Citons [Hic01] qui traite de l'évolution des applications et des transformations, [CH06] qui propose de la génération de code, [DT03] qui traite de la réecriture automatique des requêtes, et [AS06] qui, via ses "Access Program Update Mechanics", propose de bonnes pratiques à mettre en oeuvre en fonction de la nature de la transformation rencontrée. Ces bonnes pratiques sont illustrées par des exemples, elles ne sont pas formalisées, il est donc difficile de se baser sur cet ouvrage pour les automatiser. Enfin, citons [CMZ08] qui propose PRISM, un puissant outil de gestion d'adaptation des programmes à l'évolution des bases de données relationnelles. Malheureusement, PRISM se présente comme une boite noire dont seuls les effets sont visibles. Il est donc difficile de connaitre les mécanismes sous-jacents et les règles de conversion et de génération utilisées.

Notre intérêt se situe au niveau du lien entre les aspects "*modification du schéma*" et "*adaptation des programmes*". Nous souhaitons traiter ce lien en utilisant le mécanisme des vues relationnelles.

Parmis les différentes techniques permettant de réduire l'effort d'adaptation des programmes, deux se basent sur le mécanisme des vues relationnelles :

A) {***backward views***} - consiste à utiliser une ou plusieurs vues pour permettre aux programmes d'accéder aux nouvelles structures de données de manière transparente, c'est-à-dire sans (trop de) modifications. Autrement dit, une vue "*backward*" définit l'ancienne structure $\mathcal{S}_{t-1}$ comme une vue de la nouvelle structure $\mathcal{S}_t$.

B) {***forward views***} - consiste à conserver la structure du schéma d'origine et à utiliser une ou plusieurs vues de manière à simuler les transformations de la base de données. Autrement dit, une vue "*forward*" définit la nouvelle structure $\mathcal{S}_t$ comme une vue de l'ancienne structure $\mathcal{S}_{t-1}$.

Objectifs

L'objectif général de ce mémoire est de fournir un outil d'aide à l'adaptation des programmes, de manière la plus intuitive et la plus transparente possible, en utilisant la technique des {***backward views***}. Notre étude se fera dans le contexte des bases de données relationnelles. Nous ferons l'hypothèse que la base de données est modifiée, que les données sont migrées, et que tous les problèmes liés à ces deux apsects sont réglés. Nous allons donc tenter de minimiser la tâche de l'administrateur qui consiste à adapter les programmes qui accédaient à la base de données avant sa transformation. Pour ce

faire, nous proposerons une méthodologie, ainsi que des règles formelles de génération de vues dépendantes de la nature de la transformation traitée. Ensuite, nous implémenterons cette méthodologie en exploitant les règles que nous aurons définies, de manière à proposer un outil de génération automatique de vues pour assister le programmeur lors de l'évolution.

Plan du mémoire

Ce mémoire est divisé en quatre parties. Suite à cette introduction, il nous reste trois parties à développer :

Tout d'abord, la **Partie II - *"Concepts"*** a pour objectif de préciser certains concepts qui seront utiles pour ce mémoire :

- le **Chapitre 1 - *"Vues SQL"*** présente brièvement le concept de vue.
- le **Chapitre 2 - *"Le modèle GER"*** présente le modèle "Generic Entity Relationship", modèle plus générique que le modèle relationnel. Il nous sera utile pour la représentation des schémas et de leur transformation.
- le **Chapitre 3 - *"Transformations"*** présente le concept de transformation en détaillant celles qui feront l'objet de notre étude.

Ensuite, la **Partie III - *"Contribution"*** présente le fruit de la recherche effectuée dans le cadre de ce mémoire.

- le **Chapitre 4 - *"Approche méthodologique"*** définit une approche méthodologique qui nous permettra de générer la/les vues qui serviront à améliorer la compatibilité du code applicatif suite à l'évolution de la base de données.
- le **Chapitre 5 - *"Règles de génération de vues"*** présente des règles de génération de vues pour chacune des transformations traitées dans ce mémoire.
- le **Chapitre 6 - *"Outils"*** présente l'outil *"Générateur de vues"* développé dans le cadre de ce mémoire. Il implémente la méthodologie et les règles précédemment établies.
- le **Chapitre 7 - *"Etude de cas"*** présente une étude de cas réalisée dans le but de tester l'impact au niveau des performances suite à l'utilisation des vues générées.

Enfin, la dernière partie, **Partie IV - *"Conclusion"***, présente les conclusions ainsi que les perspectives futures de ce mémoire.

Deuxième partie

Concepts

Chapitre 1

Vues SQL

Ce chapitre a pour but de présenter brièvement le concept de vue. Après avoir défini ce qu'est une vue et cité ses différentes utilités, nous rappelerons et illustrerons la syntaxe associée. Ensuite nous nous intéresserons à certains types de vues que sont les vues actualisables et les vues matérialisées. Enfin, nous terminerons en synthétisant à l'aide d'un tableau les limites de l'implémentation actuelle des vues dans différents SGBD [1] relationnels. Ce chapitre est basé sur [Cel95, Hai00, Lit97, Ger03, Eyr08]

1.1 Définition

Une vue est une table virtuelle construite à partir d'une ou plusieurs tables réelles (de base) et/ou à partir d'autres vues [2] (Figure 1.1). Généralement, c'est la requête d'une vue qui est stockée dans le schéma et non son résultat [3], ce dernier sera généré à chaque sollicitation de la vue. Les données issues d'une vue peuvent être extraites via une requête SQL comme c'est le cas avec une table réelle.

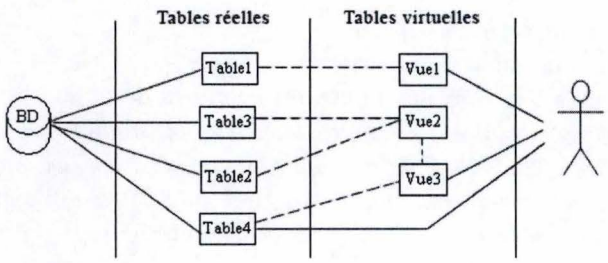


FIGURE 1.1 – *Vues*

1. **S**ystème de **G**estion de **B**ase de **D**onnées
2. On parle alors de "Vues imbriquées". Il y a deux restrictions : il ne faut pas de références récursives et il faut impérativement des tables de base sous-jacentes.
3. Il existe des vues pour lesquelles le résultat est stocké. (voir Section 1.6 - Vues matérialisées)

1.2 Intérêts et inconvénients

L'utilisation des vues présente plusieurs avantages parmis lesquels :

- **la simplification de la base de données.** Par exemple en donnant des noms plus significatifs aux colonnes, ou encore en masquant certaines colonnes inutiles.
- **la gestion plus aisée des droits d'accès** en masquant certaines informations aux utilisateurs.
- **la réutilisation d'une requête fréquemment utilisée** en la nommant, ce qui permet aussi de faciliter sa compréhension, et d'éviter des oublis en cas de modification.

Malheureusement, elles ont aussi leurs limites...

- contrairement aux tables réelles, elles ne permettent pas la création d'index.
- comme nous le verrons ci-dessous (Section 1.5), la mise à jour des données issues d'une vue est généralement impossible.

1.3 Syntaxe

Pour chaque commande (création, modification, et suppression), cette section présente les primitives associées suivant la norme SQL97.

Création - CREATE VIEW

Une vue peut-être créée via la syntaxe suivante :

```
CREATE VIEW <nom_vue> [<liste_colones>] AS
  <instruction_select>
  [WITH [CASCADED|LOCAL] CHECK OPTION]
```

où

- `nom_vue` : *le nom de la vue à créer.*
  Il doit être unique au sein du schéma.
- `liste_colonnes` : *la liste des noms des colonnes de la vue.*
  Le nombre de ces colonnes doit être identique au nombre de colonnes retournées par l'instruction `SELECT`. *S'ils ne sont pas précisés, les noms des colonnes seront ceux des colonnes résultants de l'instruction* `SELECT`. *Notons que si deux colonnes du résultat portent le même nom, il faut obligatoirement préciser le nom des colonnes de la vue pour lever l'ambiguité.*
- `<instruction_select>` : *l'instruction* `SELECT` *définissant la vue.*
- `WITH [CASCADED|LOCAL] CHECK OPTION`
  Cette clause est une contrainte qui garantit que seules les lignes du type de celles référencées par la vue pourront faire l'objet d'un ajout ou d'une mise à jour.
  Le mot-clé `CASCADED` *(valeur par défaut) étend la portée de la clause à toutes les vues sous-jacentes, le mot-clé* `LOCAL` *restreint la portée de la clause à la vue.*

Modification - ALTER VIEW

Une vue peut-être modifiée via la syntaxe suivante :

```
ALTER VIEW <nom_vue>
```

où

- `nom_vue` : *le nom de la vue à modifier.*

Suppression - DROP VIEW

Une vue peut-être supprimée via la syntaxe suivante :

```
DROP VIEW <nom_vue> [CASCADED|RESTRICT]
```

où

- `nom_vue` : *le nom de la vue à supprimer.*
- `[CASCADED|RESTRICT]` : *le mot-clé* `CASCADED` *(valeur par défaut) automatise la suppression de tous les objets dépendants de la vue (exemple : une autre vue), et le mot-clé* `LOCAL` *restreint la portée de la suppression à la vue.*

Contrairement à la suppression d'une table réelle, la suppression d'une vue ne supprime pas les données référencées par la vue.

1.4 Illustration

Nous allons à présent illustrer le concept de vue sur base d'un exemple concret.

Supposons une base de données comprenant deux tables : une table `EMPLOYE` dont chaque ligne contient les informations relatives à un employé de l'entreprise et une table `SERVICE` qui fait correspondre un numéro à nom de service.

table EMPLOYE

NEMP	NOM	PRENOM	DATENAISS	NSER
1	FERARD	Josianne	25/02/1973	3
2	COLIN	Maurice	09/01/1967	1
3	GILLET	Philippe	17/06/1977	2
5	DURANT	Sabine	27/03/1980	2

table SERVICE

NSER	NOM
1	Financier
2	Clientele
3	Juridique

Supposons que l'on souhaite déterminer quel employé travaille dans quel service, que la date de naissance, considérée comme confidentielle, ne doit pas apparaître aux yeux de tous, et enfin que la correspondance des services et de leur numéro n'est pas connue des employés.

Pour répondre au besoin de cet utilisateur, nous allons lui créer la vue AFFECTATION définie comme suit :

```
create view AFFECTATION (NOM_EMP, PRENOM_EMP, NOM_SERV) as
  select EMPLOYE.NOM, PRENOM, SERVICE.NOM
  from EMPLOYE, SERVICE
  where EMPLOYE.NSER=SERVICE.NSER
```

Une fois la vue créée, l'accès aux tables EMPLOYE et SERVICE peut se faire via la table virtuelle AFFECTATION (Figure 1.2).

```
select * from AFFECTATION
+---------+------------+-----------+
| NOM_EMP | PRENOM_EMP | NOM_SERV  |
+---------+------------+-----------+
| COLIN   | Maurice    | Financier |
| GILLET  | Philippe   | Clientele |
| DURANT  | Sabine     | Clientele |
| FERARD  | Josianne   | Juridique |
+---------+------------+-----------+
```

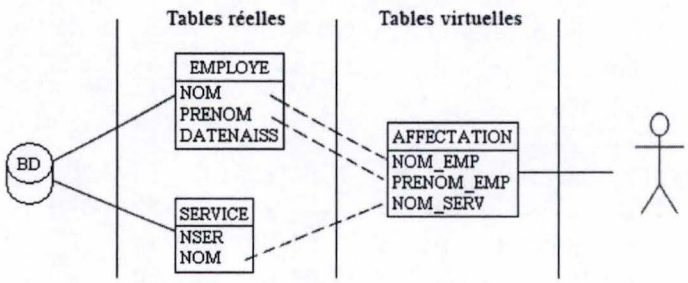


FIGURE 1.2 – *Création d'une vue* AFFECTATION

Nous pouvons modifier la vue via la commande ALTER VIEW.
Supposons, par exemple, que le prénom de l'employé ne nous intéresse plus. Nous pouvons alors modifier la vue AFFECTATION comme suit :

```
alter view AFFECTATION (nom_emp, nom_serv) as
  select EMPLOYE.NOM, SERVICE.NOM
  from EMPLOYE, SERVICE
  where EMPLOYE.NSER=SERVICE.NSER
```

L'interrogation de la vue donne alors le résultat :

```
select * from AFFECTATION
+---------+-----------+
| nom_emp | nom_serv  |
+---------+-----------+
| COLIN   | Financier |
| GILLET  | Clientele |
| DURANT  | Clientele |
| FERARD  | Juridique |
+---------+-----------+
```

Enfin, lorsque la vue ne sera plus utile, nous pourrons la supprimer à l'aide la syntaxe `DROP VIEW` :

```
drop view AFFECTATION
Query OK
```

La vue `AFFECTATION` est maintenant supprimée.

Notons que seule la vue a été supprimée, pas les données qu'elle référençait dans les tables de base :

```
select EMPLOYE.NOM, SERVICE.NOM
from EMPLOYE, SERVICE
where EMPLOYE.NSER=SERVICE.NSER
+--------+-----------+
| NOM    | NOM       |
+--------+-----------+
| COLIN  | Financier |
| GILLET | Clientele |
| DURANT | Clientele |
| FERARD | Juridique |
+--------+-----------+
```

1.5 Vues actualisables

Généralement, une vue est accessible uniquement en lecture. La mise à jour des données d'une vue est rarement possible.

Cette mise à jour nécessite...

- qu'il exite une correspondance ligne à ligne entre les colonnes de la vue et celles de la table de base.
- que l'instruction `SELECT` définissant la vue ne contienne pas de requêtes imbriquées.
- que l'instruction `SELECT` définissant la vue ne contienne pas de fonctions d'agrégation.
- que toutes les colonnes de la table de base n'ayant pas de valeur par défaut soient contenues dans la vue.

Plus précisément, la mise à jour d'une vue est impossible lorsque la vue :

- est construite à partir d'attributs dont aucun n'est identifiant.
- contient un `DISTINCT` dans sa clause `SELECT`.
- référence plus d'une table dans sa clause `FROM`.
- est construite sur base d'un opérateur ensembliste (`UNION`, ...).
- référence une vue non modifiable dans sa clause `FROM`.
- contient une clause `ORDER BY, GROUP BY, HAVING`.

La mise à jour d'une vue entraîne la mise à jour des données de la table de base sous-jacente.

1.6 Vues matérialisées

Les vues matérialisées sont des vues particulières. En effet, contrairement aux vues classiques, le résultat de l'interprétation de la vue est stocké. Ces vues sont donc des tables physiques réelles.

Suite à la création de la vue matérialisée, les données des tables de base sont dupliquées dans la nouvelle table. Ces données peuvent être mises à jour à la demande, à intervalles réguliers ou bien automatiquement lors de la mise à jour des données des tables de base. Pour plus de détails concernant les vues matérialisées, nous renvoyons à la documentation du SGBD utilisé.

Le grand avantage des vues matérialisées se situe au niveau du temps d'accès. En effet, le résultat issu d'une telle vue n'est pas regénéré à chaque appel de la vue, il est stocké dans une table réelle, c'est particulièrement intéressant lorsque la vue référence plusieurs tables. De plus, étant des tables de base, il est possible de leur associer des index. Tout ceci a évidemment un coût, le fait de créer une nouvelle table réelle nécessite un espace de stockage supplémentaire, amène de la redondance, et ajoute un surcoût à la maintenance.

1.7 Vues dans les SGBD

Tous les SGBD n'implémentent pas la norme SQL concernant les vues. A l'aide du tableau ci-dessous, nous mettons en évidence uniquement les paramètres des différentes syntaxes qui semblent utiles à notre étude, pour plus de détails veuillez consulter la documentation des différents SGBD : Oracle 10g Express edition [Ora09], MySQL 5.1 [AB08], PostgreSQL 8.3 [Gro08], Firebird 2.0 [Sof09], DB2 [IBM09].

	CREATE VIEW	ALTER VIEW	DROP VIEW
Oracle 10g	✓	✓	✓
MySQL 5.1	✓	✓	✓
PostgreSQL 8.3	✓	✓	✓
Firebird 2.0	✓	✓	✓
DB2 9.1	✓	✓	✓

	Vues materialisées	Vues actualisables
Oracle 10g	✓	✓
MySQL 5.1	✓	✓
PostgreSQL 8.3	X	X
Firebird 2.0	✓	✓
DB2 9.1	✓	✓

Chapitre 2

Le modèle GER - "Generic Entity Relationship"

Notre approche se limite aux schémas relationnels. Cependant, pour la représentation des schémas et leur transformation, nous allons nous baser sur un modèle plus générique : le modéle GER - "Generic Entity Relationship". Ce chapitre a pour but de présenter les concepts de ce modèle. Pour ce faire, nous nous baserons sur [Hai89] et [Hai02].

2.1 Définition

Le modèle GER est un modèle qui étend le modèle Entité-Association.

La stratégie standard de conception d'une base de données illustrée à la Figure 2.1 passe par trois grandes étapes : l'analyse conceptuelle, la conception logique, et la conception physique. Chacune de ces étapes, basées sur un modèle, se concrétise par respectivement un schéma conceptuel, un schéma logique, et un schéma physique. Détaillons chacun de ces schémas...

2.2 Le schéma conceptuel

Le schéma conceptuel d'une base de données est la formalisation d'un domaine d'application. Indépendant de la technologie utilisée, il peut être traduit en une multitude de schémas logiques.

Un schéma conceptuel (Entité-Association) se compose principalement de types d'entités, d'attributs, de types d'associations et de rôles.

Un **type d'entités** est une classe d'objets de même catégorie sur lequel on veut enregistrer de l'information et que l'on perçoit comme un tout (exemple : un client).

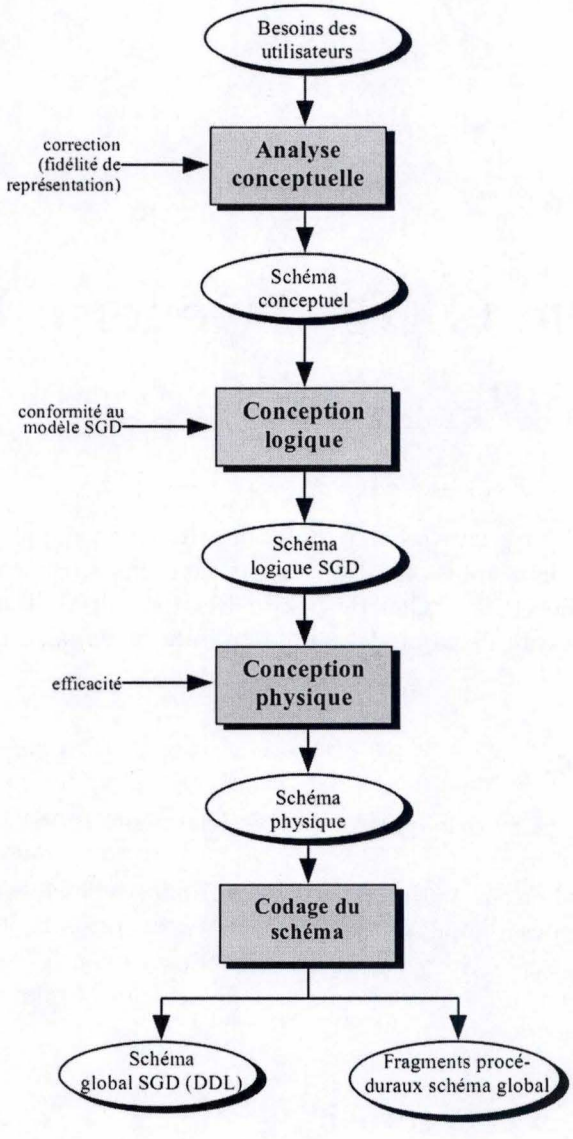


FIGURE 2.1 – *Une stratégie standard pour la conception de bases de données [Hai02]*

Les types d'entités sont liés les uns aux autres par des **types d'associations** dans lesquels ils jouent un **rôle**. Généralement, ces associations se font entre deux types d'entités (binaires), parfois trois (ternaires), et rarement plus (n-aires). En cas d'ambiguité, on peut nommer le rôle. Chaque rôle a une cardinalité qui indique le nombre [min-max] d'associations dans lesquelles une entité peut jouer un rôle.

Une propriété d'un type d'entités ou d'un type d'associations est appelée **attribut** (exemple : le nom d'un client). Chaque attribut est caractérisé par sa cardinalité qui indique le nombre [min-max] de valeurs que peut prendre l'attribut pour une instance de son parent. Un attribut est :

- atomique/composé : il est composé si sa valeur est constituée de valeurs plus élémentaires, il est atomique sinon.
- monovalué/multivalué : il est multivalué [X-N] s'il peut prendre plusieurs valeurs pour chaque instance de son parent, il est monovalué sinon [X-1].
- obligatoire/facultatif : s'il est obligatoire [1-Y], on doit lui associer une valeur pour chacune des instances de son parent, il est facultatif sinon [0-Y].
- stable/modifiable : s'il est stable, sa valeur ne peut varier au cours du temps, il est modifiable sinon.

Par défaut, un attribut est atomique-monovalué-obligatoire ce qui se traduit en partie par une cardinalité [1-1], et qui, par souci de lisibilité, n'est pas précisée sur le schéma.

La plupart du temps, un type d'entités ou un type d'associations est identifié par un ensemble d'attributs et/ou de rôles dont la valeur permet d'itentifier univoquement une instance de ce type d'entités ou de ce type d'associations. Cet ensemble se nomme un **identifiant** (*id*).

Ces différents concepts sont illustrés dans un exemple élémentaire à la Figure 2.2.

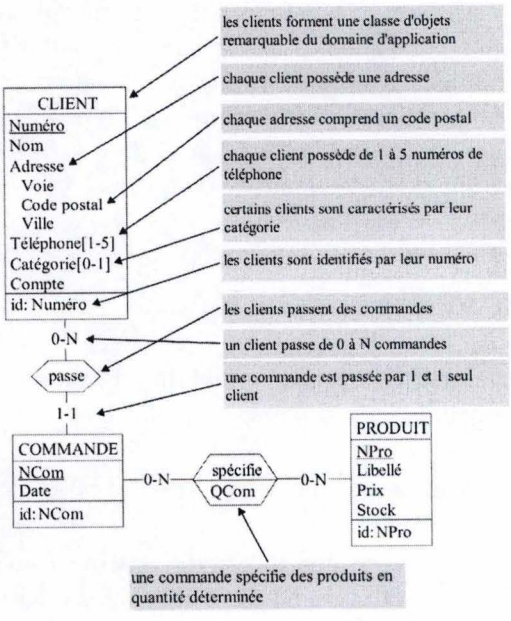


FIGURE 2.2 – *Un exemple élémentaire de schéma conceptuel [Hai02]*

En plus des concepts vus ci-dessus le schéma peut soumettre les données à des règles appelées **contraintes d'intégrité** :

- les contraintes de domaine de définition (exemple : la valeur d'un attribut de type date doit être une date).
- les identifiants impliquent une contrainte d'unicité.
- les contraintes de cardinalité (attributs et rôles) impliquent le respect des bornes [min,max].
- les contraintes de coexistence (*coex*) expriment le fait que l'existence de l'un des composants de la contrainte implique l'existence de l'autre et inversement.
- les contraintes d'exclusion (*excl*) impliquent que si un des composants de la contrainte existe, l'autre n'existe pas. Les deux composants peuvent ne pas exister en même temps.
- les contraintes au-moins-un (*at-lst-1*) expriment le fait qu'au moins un des composants de la contrainte doit exister.
- les contraintes exactement-un (*exact-1*) expriment le fait qu'un et un seul composant de la contrainte peut exister.
- les contraintes d'inclusion de rôles (incl/gr) expriment le fait que le rôle soumis à la contrainte "incl" ne peut exister que si le rôle soumis à la contrainte "gr" existe.
- les contraintes d'exclusion de rôles (r-excl/r-excl) expriment le fait que les deux rôles soumis à la contrainte ne peuvent exister en même temps.

Comme nous l'avons déjà présenté à la Figure 2.1, la conception d'une base de données passe par trois étapes que nous allons illustrer par les versions successives d'un schéma au cours de l'application de la stratégie de conception. La première étape consiste donc à présenter le schéma conceptuel de notre exemple (Figure 2.3), qui traduit un domaine d'application dans lequel il y a deux types de personnes : les clients et les fournisseurs. Un client commandant une certaine quantité d'articles à un fournisseur.

2.3 Le schéma logique (relationnel)

"Le schéma logique de données est la traduction du schéma conceptuel en structures de données propres à une famille de technologies" [Hai02].

Dans notre travail, nous considérons la technologie des bases de données relationnelles.

Le schéma logique relationnel se différencie du schéma conceptuel par le nom des concepts qu'il manipule et par la façon dont ils sont associés. On parlera généralement de type d'enregistrements, d'associations, de champs, d'identifiants, de champs de référence, et autres contraintes d'intégrité. Le tableau ci-dessous présente les correspondances :

Modèle logique	Terminologie relationnelle	Modèle conceptuel
type d'enregistrement	table	type d'entités
champ	colonne	attribut
id. primaire obligatoire	primary key	identifiant
id. secondaire obligatoire	unique	identifiant
champ de référence	clé étrangère	association

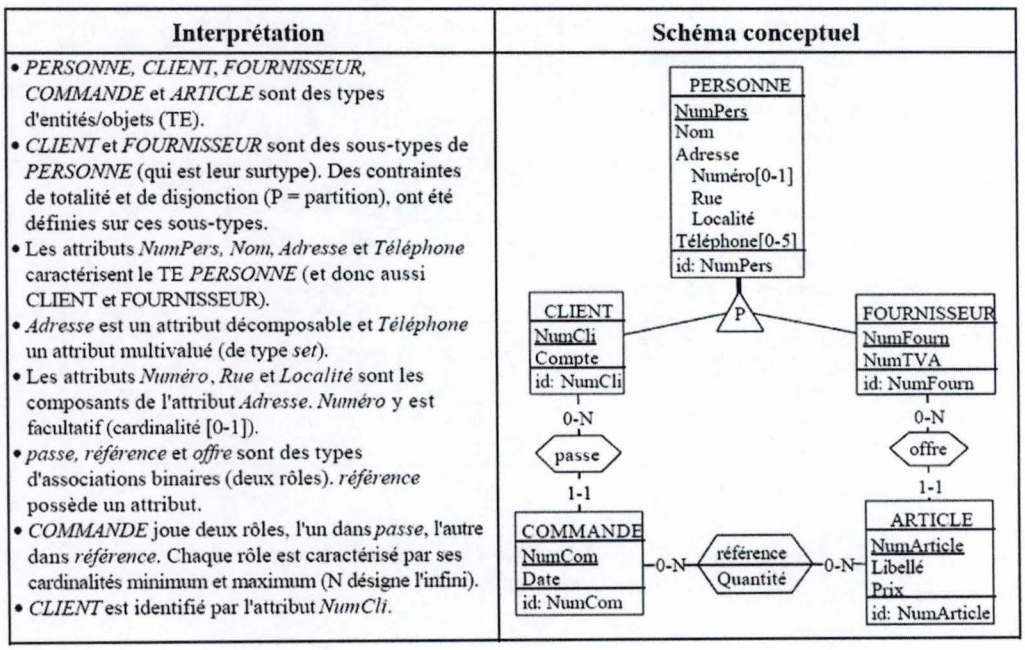


FIGURE 2.3 – *Vue graphique et interprétation d'un schéma conceptuel typique [HHE+99]*

D'autre part, certaines constructions du modèle GER ne sont pas valides dans un schéma relationnel, en voici la liste :

- relations IS-A (notion de spécialisation d'un type d'entités)
- type d'enregistrements sans champs
- champs composés
- champs multivalués
- type d'associations
- contraintes autres que les identifiants et les contraintes référencielles

Précisons le concept de "*Champ de référence*" (*ref*) : Dans le cas des bases de données relationnelles, nous parlerons plutôt de "*Clé étrangère*". Une clé étrangère est un groupe de colonnes qui référence une instance d'une autre table. Une clé étrangère peut-être accompagnée d'une contrainte d'équivalence (*equ*), qui signifie qu'à toute instance de la table référençant, il existe une instance de la table référencée, et inversémment.

La Figure 2.4 présente la traduction en schéma logique relationnel du schéma conceptuel illustré à la Figure 2.3.

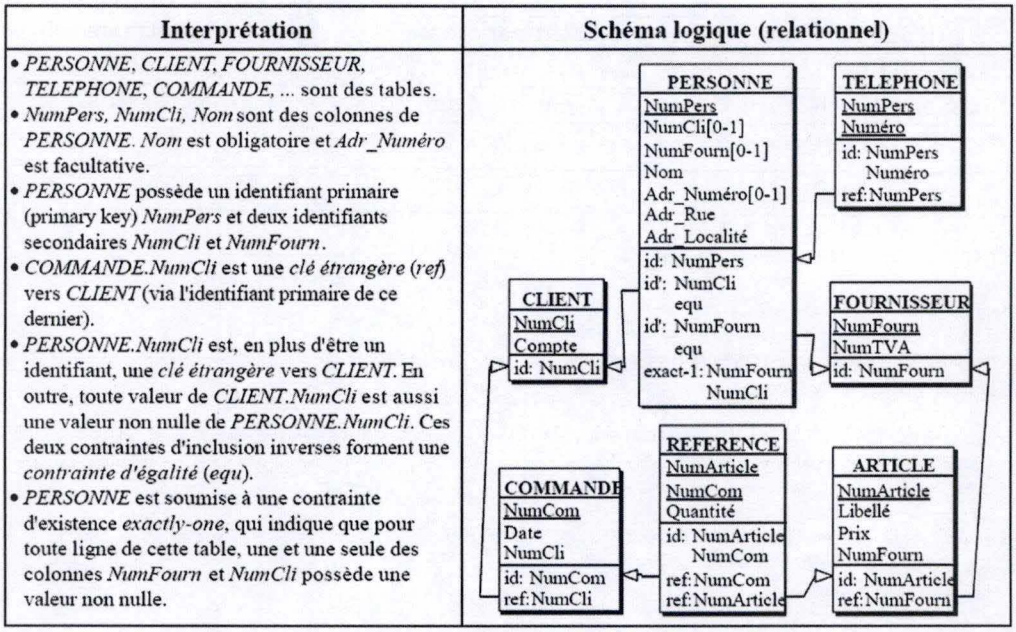


FIGURE 2.4 – *Vue graphique et interprétation d'un schéma logique relationnel [HHE+99] (traduction du schéma conceptuel de la Figure 2.3)*

2.4 Le schéma physique

Le schéma physique est l'interprétation du schéma logique dans un SGBD particulier de la famille de technologie du schéma logique (Oracle, MySQL,...). Il est agrémenté d'espaces de stockage représenté sur le schéma par des cylindres, de clés d'accès (index) représentée par l'abréviation *acc*, ainsi que de divers autres paramètres techniques.

La Figure 2.5 présente la traduction en schéma physique du schéma logique illustré à la Figure 2.4.

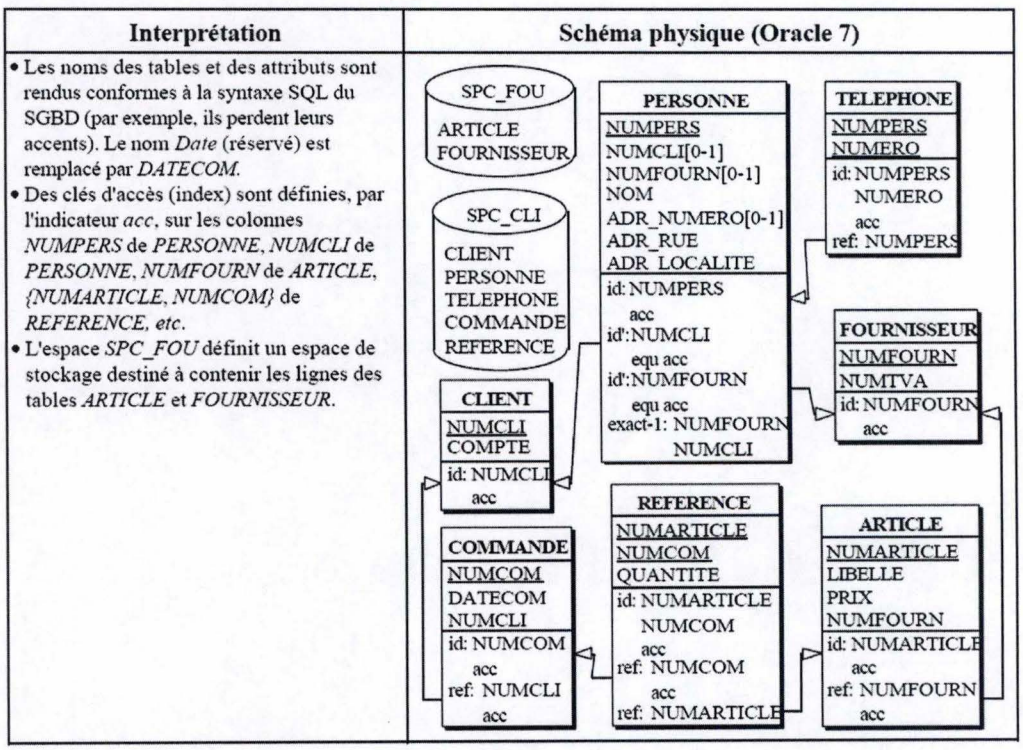


FIGURE 2.5 – *Vue graphique et interprétation d'un schéma physique relationnel [HHE+99] (traduction du schéma logique de la Figure 2.4)*

Chapitre 3

Transformations

Ce chapitre décrit le concept de transformation de schéma. Nous allons y définir la notion de transformation et nous préciserons ce qu'est une transformation réversible. Ensuite, nous présenterons un ensemble de transformations élémentaires, ainsi que le concept de transformation composée.
Pour terminer ce chapitre, nous détaillerons les étapes nécessaires à la réalisation des transformations de schéma relationnel qui feront l'objet de notre étude.
Ce chapitre se base principalement sur l'Annexe E de [Hai09].

3.1 Définition

Au cours de sa vie, une base de données est amenée à évoluer. En effet, les besoins des utilisateurs changent ce qui se traduit par la modification de la structure de la base de données, c'est à dire l'ajout, la suppression, ou la modification d'un ensemble d'objets. On parlera de transformation de schéma.

De manière générale, une transformation `T` est un opérateur qui remplace un ensemble d'objets `C` d'un schéma par un nouvel ensemble d'objets `C'`. On peut écrire : `C'=T(C)`
`C` désigne donc un ensemble d'objets qui constituent une classe de construction. Pour que la définition de transformation soit complète, il faut aussi inclure la notion de transformation des instances `c` par l'opérateur `t`. La Figure 3.1 illustre ces notions.

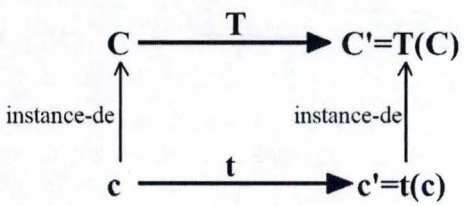


FIGURE 3.1 – *Schéma d'une transformation [HHE+99]*

3.2 Transformations réversibles

Il existe trois types de transformations de schéma :

- T+ - celles qui ***augmentent*** la sémantique du schéma (ex : ajouter un attribut).
- T- - celles qui ***diminuent*** la sémantique du schéma (ex : supprimer un attribut).
- T= - celles qui ***préservent*** la sémantique du schéma (ex : renommer un attribut).

Les transformations de types T= sont dites ***réversibles*** ou ***à sémantique constante***. Le schéma résultant d'un transformation réversible décrit le même domaine d'application que le schéma dont il est issu, c'est juste une autre représentation.
On peut écrire : `C'=T(C)=T(T'(C'))` et `C=T'(C')=T'(T(C))`

La Figure 3.2 présente un exemple de transformation réversible dans laquelle l'attribut NumTel du type d'entités `CLIENT` est transformé en un type d'entités `TELEPHONE`.

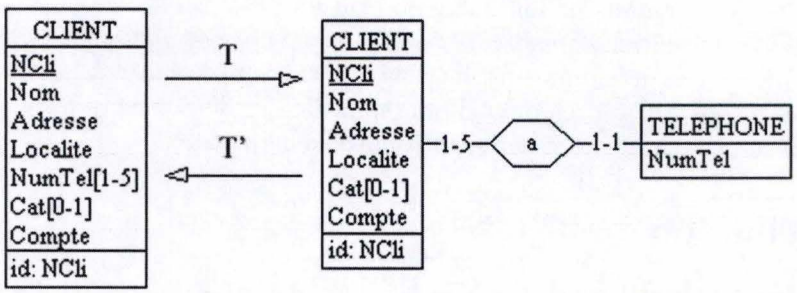


FIGURE 3.2 – *Exemple de transformation réversible - Tranformation d'un attribut en type d'entités*

Le tableau présenté à la Figure 3.3 présente une classification des principales transformations réversibles par type d'objets. Seules quelques unes d'entre elles seront étudiées dans ce mémoire.

Type d'objets	T=
Type d'entités	Modification du nom, éclatement, fusion
Relation IS-A	Transformation en type d'associations + transformation en attribut de référence
Type d'associations	Transformation en type d'entités + transformation en attribut de référence, transformation en attribut de référence
Attribut	Modification du nom, transformation en type d'entités + transformation en attribut de référence, désagrégation d'un attribut décomposable, instantiation d'un attribut multivalué en attributs mono-valués, concaténation d'un attribut multivalué

FIGURE 3.3 – *Classification des principales transformations réversibles par type d'objet [HHE+99]*

3.3 Transformations élémentaires

Avant de présenter les transformations qui feront l'objet de ce mémoire, il est nécessaire de décrire certaines transformations élémentaires. Ces transformations sont définies sur le modèle GER présenté au Chapitre 2. Les intitulés de ces transformations ainsi que les images descriptives sont toutes extraites de l'Annexe E de [Hai09].

3.3.1 Transformation d'un attribut atomique monovalué en type d'entités par représentation des instances

Dans le cas de la représentation des instances, l'instance du nouveau type d'entités n'existe que s'il existe une instance du type d'associations le liant à une instance du type d'entités auquel il appartenait avant la transformation. L'attribut à décomposer peut être ou non identifiant, s'il l'est, il devriendra un identifiant du nouveau type d'entités. (Figure 3.4)

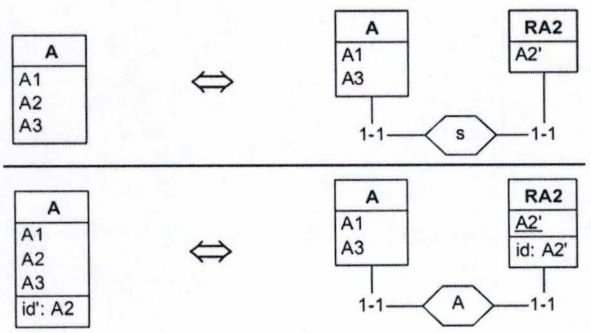


FIGURE 3.4 – *Transformation d'un attribut atomique monovalué en type d'entités par représentation des instances [Hai09]*

3.3.2 Décomposition d'un attribut atomique monovalué en attribut composé

Cette transformation consiste à décomposer un attribut en sous-attributs qui peuvent être de tailles ou de types différents de ceux de l'attribut de base. Dans le cas illustré à la Figure 3.5, suite à la transformation, l'attribut `A2` est composé de deux sous-attributs `A21` et `A22`.

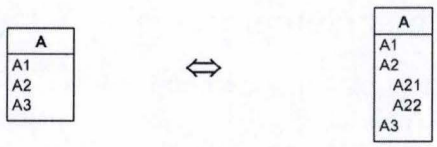


FIGURE 3.5 – *Décomposition d'un attribut atomique monovalué en attribut composé*

3.3.3 Désagrégation d'un attribut composé monovalué

Cette transformation consiste à remplacer l'attribut composé par ses composants. Dans la Figure 3.6, l'attribut `A2` composé des sous-attributs `A21` et `A22` est désagrégé, c'est à dire que les sous-attributs remontent d'un niveau et prennent la place de l'attribut dont ils sont issus. Par souci de clarté, on peut les préfixer avec `A2`, le nom de l'ancien attribut parent.

FIGURE 3.6 – *Désagrégation d'un attribut composé monovalué [Hai09]*

3.3.4 Transformation d'un attribut composé monovalué en type d'entités par représentation des instances

Cette transformation est identique à la transformation d'un attribut atomique monovalué en type d'entités par représentation des instances présentée ci-dessus. Si ce n'est qu'en plus, il semble logique de désagréger l'attribut transformé. (Figure 3.7)

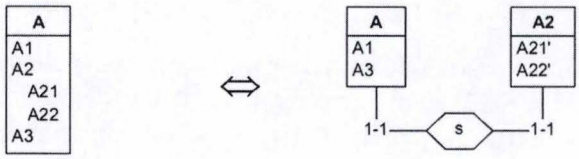


FIGURE 3.7 – *Transformation d'un attribut composé monovalué en type d'entités par représentation des instances [Hai09]*

3.3.5 Agrégation d'un groupe d'attributs

Cette transformation consiste à rendre les membres d'un groupe d'attributs composants d'un nouvel attribut composé. (Figure 3.8)

FIGURE 3.8 – *Agrégation d'un groupe d'attributs [Hai09]*

3.3.6 Transformation d'attributs en type d'associations

Cette transformation, surtout utile en rétro-ingénierie, consiste à transformer un groupe d'attributs constituant une clé étrangère en un type d'associations. (Figure 3.9)

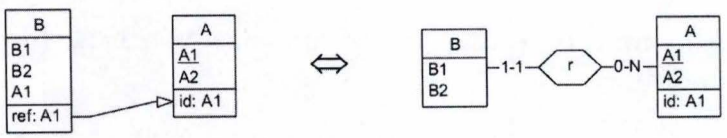


FIGURE 3.9 – *Transformation d'attributs en type d'associations [Hai09]*

3.3.7 Décomposition d'un type d'entités par partitionnement vertical

Cette transformation consiste à déplacer un ensemble d'attributs et/ou de rôles d'un type d'entités et à les réunir dans un nouveau type d'entités. Le nouveau type d'entités ainsi créé est relié au type d'entités faisant l'objet de la transformation par un type d'associations 1 :1. (Figure 3.10)

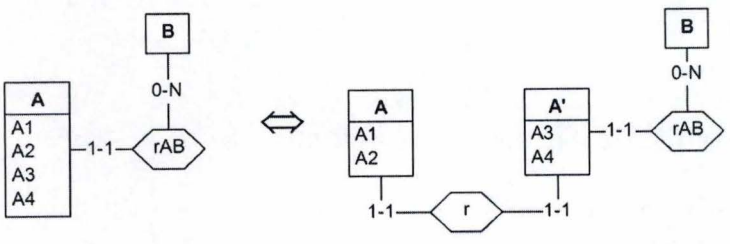


FIGURE 3.10 – *Décomposition d'un type d'entités par partitionnement vertical [Hai09]*

3.3.8 Fusion bijective de types d'entités

Transformation applicable uniquement à deux types d'entités reliés entre eux par un type d'associations 1 :1, elle consiste à migrer les composants d'un des deux types d'entités (l'absorbé) vers l'autre (l'absorbant). Dans la Figure 3.11, les types d'entités `A1` (l'absorbant) et `A2` (l'absorbé) sont fusionnés, il en résulte la table `A`.

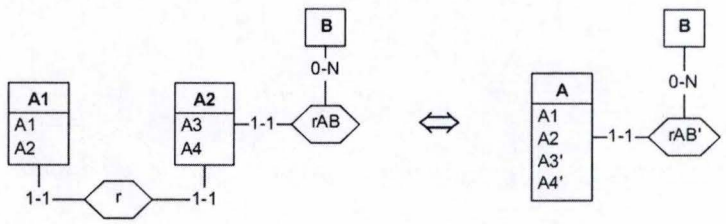


FIGURE 3.11 – *Fusion bijective de types d'entités [Hai09]*

3.3.9 Transformation d'un type d'associations 1 :1 binaire en clé étrangère

"Un type d'associations 1 :1 se transforme en une clé étrangère monovaluée identifiante." [Hai09] (Figure 3.12)

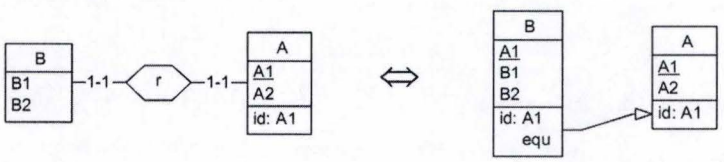


FIGURE 3.12 – *Transformation d'un type d'associations 1 :1 binaire en clé étrangère [Hai09]*

3.4 Transformations composées

Une transformation composée est un transformation constituée d'une chaîne de transformations élémentaires.

La transformation composée $T_{12} = T_2 \circ T_1$ est obtenue en appliquant la transformation T_2 sur le résultat obtenu suite à l'application de la transformation T_1.

3.5 Description des transformations qui feront l'objet de notre étude

Dans le cadre de ce mémoire, nous partons d'un schéma relationnel source auquel on applique une transformation réversible T, il en résulte un schéma relationnel cible. La transformation T s'effectue dans le modèle relationnel et aboutit dans le modèle relationnel (Rel-to-Rel). T est en fait une *macro*-transformation qui est composée d'une ou plusieurs transformations élémentaires. Nous allons quitter le modèle relationnel pour passer dans le modèle GER le temps de décrire chacune des étapes qui se cachent derrière la macro-transformation T. (Figure 3.13)

$T \equiv$ macro-transformation (Rel-to-Rel).
$T_1 ... T_n \equiv$ transformations élémentaires du modèle GER (GER-to-GER).

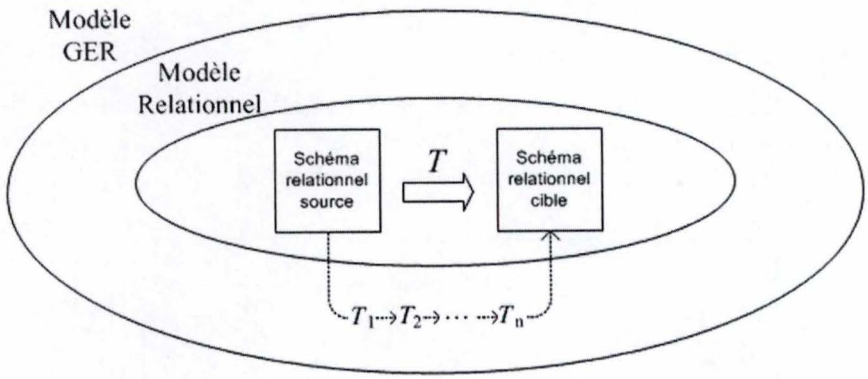


FIGURE 3.13 – *Décomposition d'un macro-transformation Rel-to-Rel*

Dans la suite de cette section, nous allons détailler la composition de chacune des transformations Rel-to-Rel que nous avons choisi d'étudier. Pour ce faire, nous illustrerons le chemin de transformation parcouru et, à chaque étape, nous donnerons l'intitulé de la transformation élémentaire GER rencontrée.

3.5.1 Renommer une table

Le chemin de transformation illustré à la Figure 3.14 est celui emprunté par le renommage de la la table `T` en `W`, il passe par :
1 - le renommage du type d'entités `T` en `W`

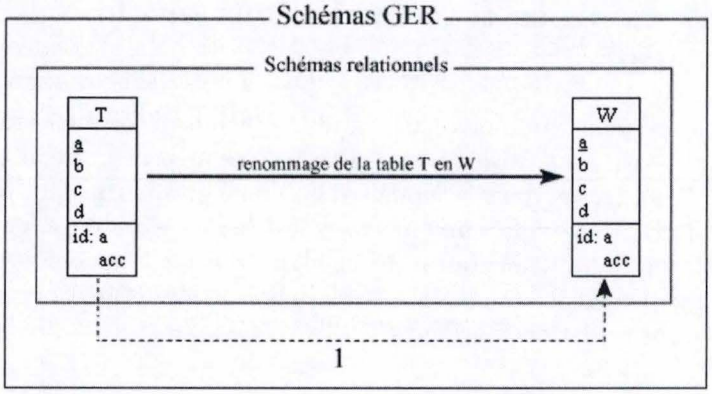


FIGURE 3.14 – *Chemin de transformation pour le renommage d'une table*

3.5.2 Renommer une colonne

Le chemin de transformation illustré à la Figure 3.15 est celui emprunté par le renommage de la colonne `b` de la table `T` en `x`, il passe par :
1 - le renomage de l'attribut `b` du type d'entités `T` en `x`

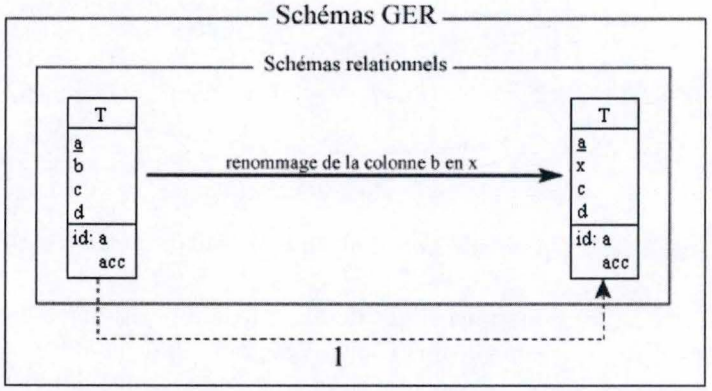


FIGURE 3.15 – *Chemin de transformation pour le renommage d'une colonne*

3.5.3 Scinder une colonne

Supposons que l'on scinde la colonne `b` de la table `T` en deux colonnes `b1` et `b2`. Le chemin de transformation associé à cette transformation est illustré à la Figure 3.16, il passe par :
1 - la décomposition de l'attribut atomique monovalué `b` en attribut composé
2 - la désagrégation de l'attribut composé monovalué `b` en `b1` et `b2`

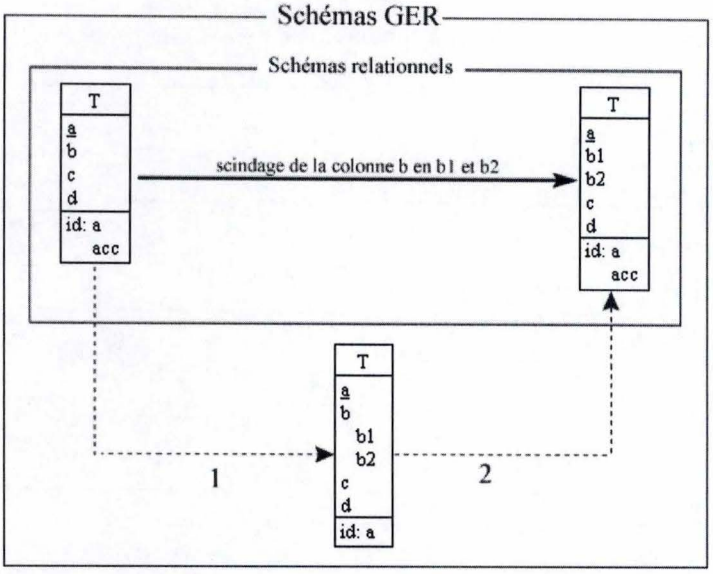


FIGURE 3.16 – *Chemin de transformation pour le scindage d'une colonne*

3.5.4 Scinder une table

Scinder une table consiste à couper une table en deux en déplacant une ou plusieurs colonnes vers une nouvelle table. Supposons que l'on scinde la table `T` en deux tables `T` et `T2`, cette dernière héritant des colonnes `b` et `c`.

Si cela ne concerne qu'une seule colonne, le chemin de transformation consiste en la transformation de l'attribut atomique monovalué en type d'entités par représentation des instances.

Sinon, un premier chemin de transformation illustré à la Figure 3.17 passe par :
1 - la décomposition du type d'entités `T` par partitionnement vertical
2 - la transformation du type d'associations `R` en clé étrangère

Ou un second chemin de transformation possible illustré à la Figure 3.18 passe par :
1 - le groupement des attributs `b` et `c` à déplacer
2 - l'agrégation du groupe d'attributs précédement créé en un attribut `T2`
3 - la transformation de l'attribut composé monovalué `T2` en type d'entités
4 - la transformation du type d'associations `R` en clé étrangère

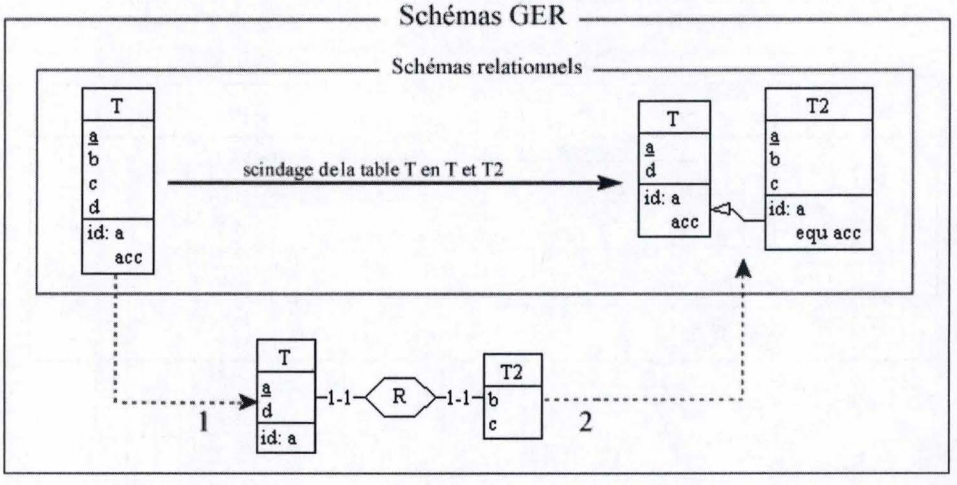


FIGURE 3.17 – *Chemin de transformation pour le scindage d'une table*

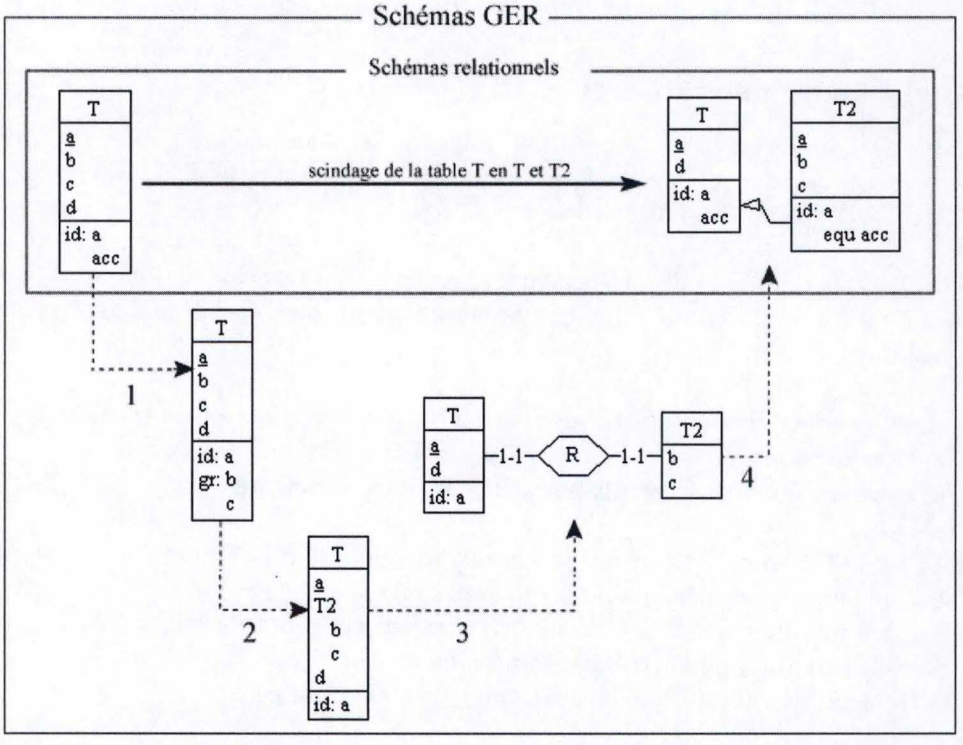


FIGURE 3.18 – *Chemin de transformation pour le scindage d'une table*

3.5.5 Fusionner deux tables

Le chemin de transformation illustré à la Figure 3.19 est celui emprunté lors de la fusion de deux tables `T` et `T2` en une table `T`, il passe par :
1 - la transformation de l'attribut `a` de `T2` en type d'associations `R`
2 - la fusion bijective des types d'entités `T` et `T2`

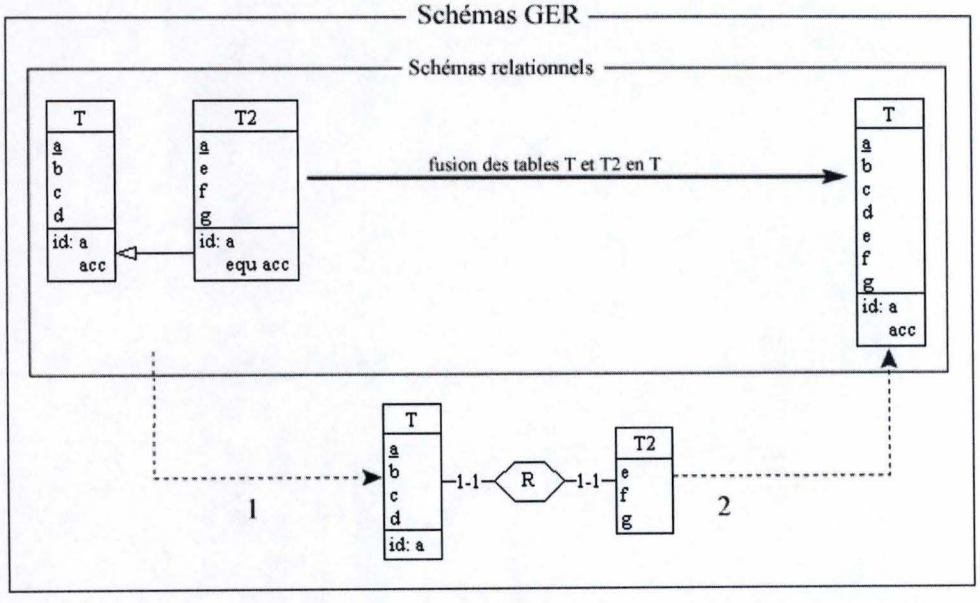


FIGURE 3.19 – *Chemin de transformation pour la fusion de deux tables*

Troisième partie

Contribution

Chapitre 4

Approche méthodologique

Ce chapitre présente l'approche méthodologique employée afin d'obtenir la vue permettant de rendre l'évolution d'un schéma relationnel la plus transparente possible pour le programme. Après la présentation de la problématique et de la solution proposée, nous détaillerons chaque étape du processus de génération de la vue.

4.1 Problématique

A l'origine, la base de données $\mathcal{BD}_0$ basée sur le schéma $\mathcal{S}_0$ est exploitée à l'aide d'un ensemble de requêtes $\mathcal{R}_0$ (Figure 4.1).

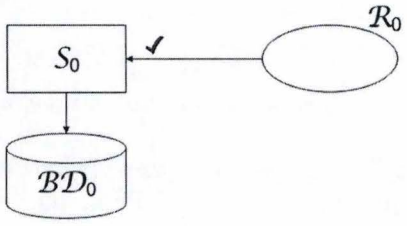


FIGURE 4.1 – *Base de données avant transformation*

Suite à la transformation $\mathtt{T}_1$ de $\mathcal{S}_0$ destinée à répondre au nouveaux besoins des utilisateurs, un nouveau schéma $\mathcal{S}_1$ est créé, il traduit l'évolution de la structure de la base de données qui devient $\mathcal{BD}_1$. Suite à cette modification de structure, jusqu'à 70% des requêtes de $\mathcal{R}_0$ peuvent échouer sur $\mathcal{S}_1$ [CMTZ08]. Nous distinguerons donc deux sous-groupes au sein du groupe $\mathcal{R}_0$; les requêtes qui, suite à la transformation, fonctionnent encore, $\mathcal{R}_0\_ok$, et celles qui ne fonctionnent plus, $\mathcal{R}_0\_ko$ (Figure 4.2).

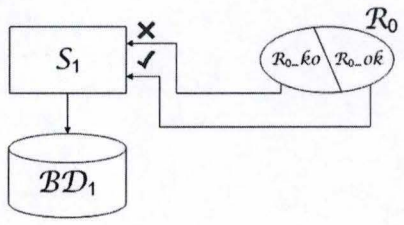


FIGURE 4.2 – *Base de données après transformation*

4.2 Solution proposée

Pour limiter le nombre requêtes qui ne fonctionnent plus, nous proposons une solution qui consiste à ajouter une vue simulant la structure de la base de données telle qu'elle était avant la transformation (Figure 4.3).

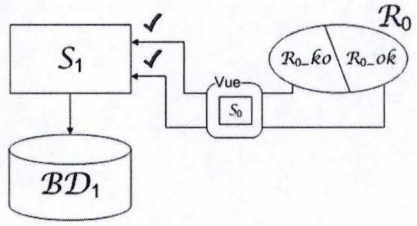


FIGURE 4.3 – *Solution proposée*

L'illustration de la solution fait l'hypothèse très optimiste que toutes les requêtes de $\mathcal{R}_0\_ko$ fonctionnent à nouveau grâce à l'ajout de la vue. En fonction de la nature des requêtes, ce ne sera évidemment pas toujours le cas. Notre but est d'améliorer au mieux la compatibilité du code applicatif existant suite à la transformation du schéma.

Une fois la solution appliquée, l'administrateur de la base de données peut agrémenter l'ensemble $\mathcal{R}_0$ de nouvelles requêtes $\mathcal{R}_1\_new$, l'ensemble des requêtes devient alors $\mathcal{R}_1$ (Figure 4.4).

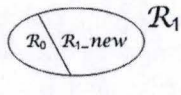


FIGURE 4.4 – *Nouvel ensemble de requêtes* $\mathcal{R}_1$

Au fil des transformations, notre solution ressemble à celle-ci (Figure 4.5) :

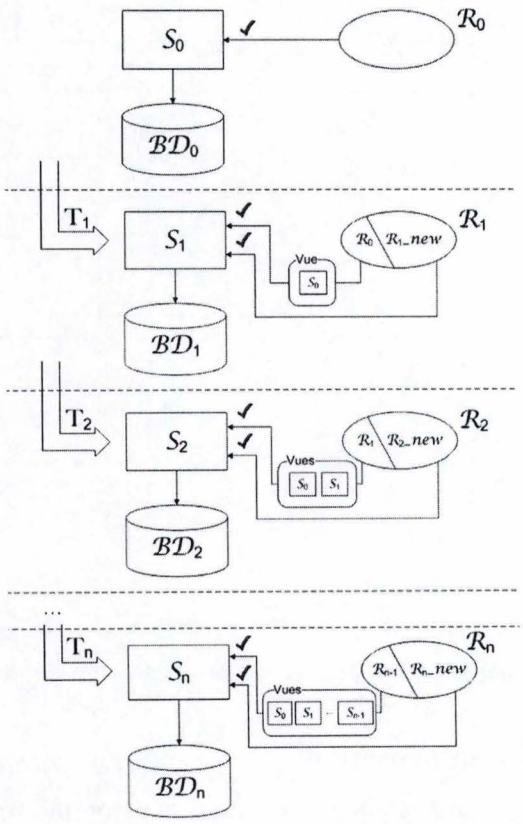


FIGURE 4.5 – *Au fil des transformations...*

4.3 Génération de la vue

La vue simulant la structure de la base de données précédant la transformation sera générée au terme du processus schématisé à la Figure 4.6. Ce processus est composé de sous-processus détaillés et illustrés ci-dessous selon l'exemple de la décomposition d'une table.

Nous ferons l'hypothèse que :

- Le mapping est initialisé sur le schéma source, $\mathcal{S}_{src}$. Les objets sont donc maintenant identifiés par un numéro unique au sein du schéma.
- Le schéma cible, $\mathcal{S}_{cbl}$, résulte de la transformation T d'une copie du schéma source, $\mathcal{S}_{src}$
- Les données ont été migrées

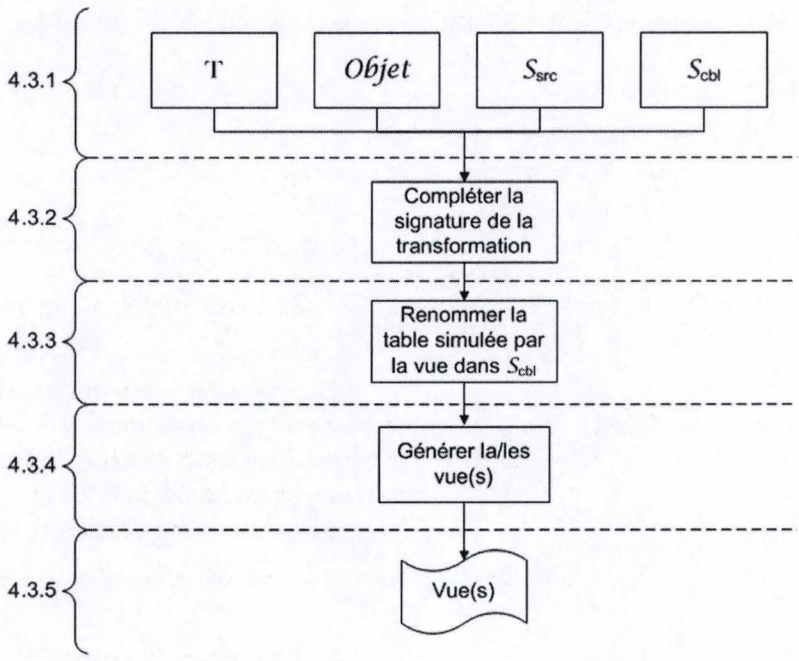


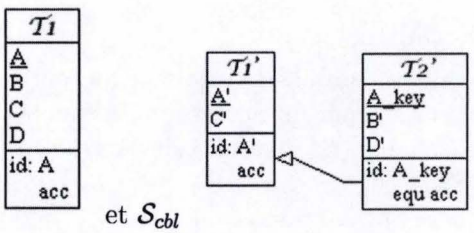
FIGURE 4.6 – *Processus de génération de vue*

4.3.1 Collecter les informations

Nous collectons un minimum d'informations qui nous permettra d'identifier la transformation, à savoir :

- T - la nature de la transformation
- *Objet* - le nom de l'objet transformé (nom de l'objet plus, éventuellement, le nom de la table à laquelle il appartient).
- $\mathcal{S}_{src}$ - le schéma avant transformation
- $\mathcal{S}_{cbl}$ - le schéma après transformation

Dans notre exemple de la décomposition d'une table, nous collectons les informations suivantes :

- T = "décomposition d'une table"
- *Objet* = "$\mathcal{T}_1$"
- Les deux schémas : $\mathcal{S}_{src}$  et $\mathcal{S}_{cbl}$

4.3.2 Compléter la signature de la transformation

Associé à la nature de la transformation et au nom de l'objet transformé, nous pouvons exploiter le mapping entre les schémas source et cible afin de retrouver l'objet transformé dans le nouveau schéma, et ainsi compléter la signature de la transformation avec les valeurs des arguments qui seraient encore inconnues.

Dans notre exemple, la génération de la vue simulant la table avant sa décomposition nécessite de connaitre le nom des nouvelles tables résultant de la décomposition, ainsi que l'affectation des colonnes à ces nouvelles tables. Nous pouvons retrouver ces informations en exploitant le mapping. Après avoir identifié la table $\mathcal{T}_1$ au sein du schéma $\mathcal{S}_{src}$, nous allons parcourir toutes ses colonnes et pour chacunes d'elles, rechercher la colonne correspondante dans le schéma $\mathcal{S}_{cbl}$. Au fil de ce parcours, nous allons alimenter différentes variables (*select, from, where*) qui nous permettrons de construire la vue.

La table de mapping réduite aux objets du schéma $\mathcal{S}_{src}$ est de la forme :

$$\begin{aligned} \mathcal{T}_1 &\rightarrow \mathcal{T}_1{}' \\ A &\rightarrow A' \\ B &\rightarrow B' \\ C &\rightarrow C' \\ D &\rightarrow D' \end{aligned}$$

où $O \rightarrow O'$ signifie que l'objet O' de $\mathcal{S}_{cbl}$ correspond à l'objet O de $\mathcal{S}_{src}$

Le tableau ci-dessous décrit l'évolution du contenu des variables *select, from, where* au fil du parcours des colonnes de $\mathcal{T}_1$ et de la table de mapping :

$table_{src}.col$	$table_{cbl}.col$	*select*	*from*	*where*
$\mathcal{T}_1.A$	$\mathcal{T}_1{}'.A'$	$\mathcal{T}_1{}'.A'$	$\mathcal{T}_1{}'$	
$\mathcal{T}_1.B$	$\mathcal{T}_2{}'.B'$	$\mathcal{T}_1{}'.A'\|\mathcal{T}_2{}'.B'$	$\mathcal{T}_1{}'\|\mathcal{T}_2{}'$	
$\mathcal{T}_1.C$	$\mathcal{T}_1{}'.C'$	$\mathcal{T}_1{}'.A'\|\mathcal{T}_2{}'.B'\|\mathcal{T}_1{}'.C'$	$\mathcal{T}_1{}'\|\mathcal{T}_2{}'$	
$\mathcal{T}_1.D$	$\mathcal{T}_2{}'.D'$	$\mathcal{T}_1{}'.A'\|\mathcal{T}_2{}'.B'\|\mathcal{T}_1{}'.C'\|\mathcal{T}_2{}'.D'$	$\mathcal{T}_1{}'\|\mathcal{T}_2{}'$	
condition de jointure		$\mathcal{T}_1{}'.A'\|\mathcal{T}_2{}'.B'\|\mathcal{T}_1{}'.C'\|\mathcal{T}_2{}'.D'$	$\mathcal{T}_1{}'\|\mathcal{T}_2{}'$	$\mathcal{T}_2{}'.A\_key = \mathcal{T}_1{}'.A'$

Détaillons la première ligne du tableau : la première colonne de la table $\mathcal{T}_1$, A, a pour correspondant la colonne A'. On traite A' en l'ajoutant à la variable *select*, nous vérifions ensuite si la table à laquelle A' appartient est déjà dans la variable *from*. Ce n'est pas le cas, nous lui ajoutons donc $\mathcal{T}_1{}'$.
On procède de la même manière pour chacune des colonnes de $\mathcal{T}_1$.
Enfin, on termine en analysant la *condition de jointure* et en l'ajoutant à la variable *where*.

4.3.3 Renommer la table simulée par la vue dans $\mathcal{S}_{cbl}$

Etant donné que le nom de la vue qui va être générée doit porter le même nom que celui de la table à laquelle elle va se substituer, et que le nom de la vue doit être unique au sein du schéma, nous devons renommer la table faisant l'objet de la transformation dans le schéma cible, $\mathcal{S}_{cbl}$.

Dans notre exemple si une des deux tables porte le même nom que $\mathcal{T}_1$, alors, il faut la renommer.
Supposons que le nom de $\mathcal{T}_1{}'$ = le nom de $\mathcal{T}_1$.
Alors $\mathcal{T}_1{}'$ est renommée $\mathcal{T}_1{}''$ et nos variables *select, from, where* deviennent :

select	*from*	*where*
$\mathcal{T}_1{}''.A' \vert \mathcal{T}_2{}'.B' \vert \mathcal{T}_1{}''.C' \vert \mathcal{T}_2{}'.D'$	$\mathcal{T}_1{}'' \vert \mathcal{T}_2{}'$	$\mathcal{T}_2{}'.A'\_key = \mathcal{T}_1{}''.A'$

4.3.4 Générer la/les vue(s)

A l'aide des règles de génération de vues définies au Chapitre 5 et des données issues des étapes précédentes, une vue portant le même nom que la table transformée dans $\mathcal{S}_{src}$ est créée.

Dans notre exemple, la vue générée sera de la forme :

`CREATE VIEW` $\mathcal{T}_1(A, B, C, D)$ `AS`
`SELECT` $\mathcal{T}_1{}''.A', \mathcal{T}_2{}'.B', \mathcal{T}_1{}''.C', \mathcal{T}_2{}'.D'$
`FROM` $\mathcal{T}_1{}'', \mathcal{T}_2{}'$
`WHERE` $\mathcal{T}_2{}'.A\_key = \mathcal{T}_1{}''.A'$

4.3.5 Vue(s)

La vue ainsi générée permet une meilleure compatibilité des anciennes requêtes avec le nouveau schéma.

Chapitre 5

Règles de génération de vues

Pour chacune des transformations décrites au Chapitre 3, nous allons associer une vue qui diminuera l'impact négatif sur la compatibilité du code applicatif. Pour ce faire, nous allons identifier les données nécessaires à la création de la vue, et nous formulerons de manière générale la syntaxe `CREATE VIEW`.

☞ remarque : Etant donné que le nom de la vue doit être unique au sein du schéma, et que pour atteindre notre objectif ce nom doit être celui de la table ayant été mofidiée, il est nécessaire de renommer la table ayant fait l'objet de la transformation, ce nouveau nom sera noté $\mathcal{T}_{cbl}$, nom de la table cible.

5.1 Renommer une table

Renommer la table $\mathcal{T}_{src}$ en $\mathcal{T}_{cbl}$

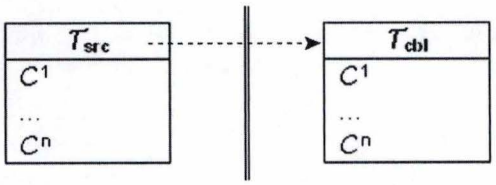


FIGURE 5.1 – *Renommer une table*

Signature de la transformation : $rename\_table(\mathcal{T}_{src}, \mathcal{T}_{cbl})$

Notations :

- $\mathcal{T}_{src}$: le nom de la table source
- $\mathcal{T}_{cbl}$: le nom de la table cible
- n : le nombre de colonnes de $\mathcal{T}_{src}$ et $\mathcal{T}_{cbl}$
- $\mathcal{C}^i_{\mathcal{T}_{src}}$: le nom de la colonne i de la table $\mathcal{T}_{src}$ où $1 \leq i \leq n$
- $\mathcal{C}^i_{\mathcal{T}_{cbl}}$: le nom de la colonne i de la table $\mathcal{T}_{cbl}$ où $1 \leq i \leq n$

Précondition :

- $\mathcal{T}_{src} \neq \mathcal{T}_{cbl}$

Alors, la vue permettant d'accéder à la table $\mathcal{T}_{cbl}$ avec le nom $\mathcal{T}_{src}$ peut-être créée comme suit :

```
CREATE VIEW T_src(C^1_T_src, ..., C^n_T_src) AS
SELECT C^1_T_cbl, ..., C^n_T_cbl
FROM T_cbl
```

5.2 Renommer une colonne

Renommer la colonne Col_{src} de la table $\mathcal{T}_{src}$ en Col_{cbl}

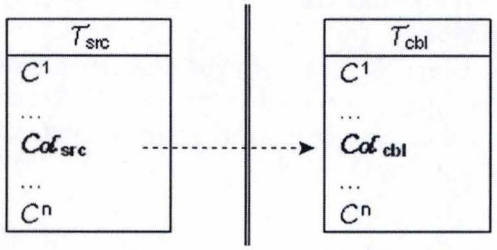


FIGURE 5.2 – *Renommer une colonne*

Signature de la transformation : $rename\_column(\mathcal{T}_{src}, Col_{src}, Col_{cbl})$

Notations :

- $\mathcal{T}_{src}$: le nom de la table source
- $\mathcal{T}_{cbl}$: le nom de la table cible
- n : le nombre de colonnes de $\mathcal{T}_{src}$ et $\mathcal{T}_{cbl}$
- $C^i_{\mathcal{T}_{src}}$: le nom de la colonne i de la table $\mathcal{T}_{src}$ où $1 \leq i \leq n$
- $C^i_{\mathcal{T}_{cbl}}$: le nom de la colonne i de la table $\mathcal{T}_{cbl}$ où $1 \leq i \leq n$
- Col_{src} : ancien nom de la colonne renommée $\wedge\ Col_{src} \in \{C^1_{\mathcal{T}_{src}}, ..., C^n_{\mathcal{T}_{src}}\}$
- Col_{cbl} : nouveau nom de la colonne renommée $\wedge\ Col_{cbl} \in \left\{C^1_{\mathcal{T}_{cbl}}, ..., C^n_{\mathcal{T}_{cbl}}\right\}$

Préconditions :

- $\mathcal{T}_{src} \neq \mathcal{T}_{cbl}$ (voir remarque page 55)
- $C^k_{\mathcal{T}_{src}} = C^k_{\mathcal{T}_{cbl}} \quad \forall k\ tq\ 1 \leq k \leq n\ \wedge\ C^k_{\mathcal{T}_{src}} \neq Col_{src}\ \wedge\ C^k_{\mathcal{T}_{cbl}} \neq Col_{cbl}$

Alors, la vue permettant d'accéder à la table $\mathcal{T}_{cbl}$ de la même manière que l'on accédait à $\mathcal{T}_{src}$ peut-être créée comme suit :

CREATE VIEW $\mathcal{T}_{src}(C^1_{\mathcal{T}_{src}}, ..., Col_{src}, ..., C^n_{\mathcal{T}_{src}})$ AS
SELECT $C^1_{\mathcal{T}_{cbl}}, ..., Col_{cbl}, ..., C^n_{\mathcal{T}_{cbl}}$
FROM $\mathcal{T}_{cbl}$

5.3 Scinder une colonne

Scinder la colonne Col_{src} en $(Col^1_{cbl}, ..., Col^m_{cbl})$

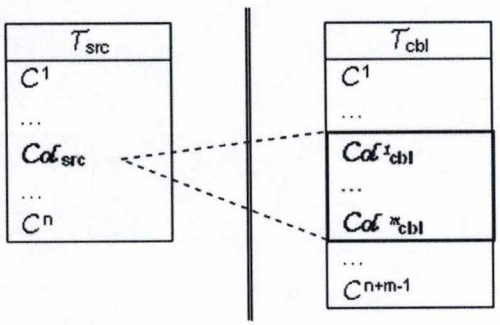


FIGURE 5.3 – *Scinder une colonne*

Signature de la transformation : $split\_column(\mathcal{T}_{src}, Col_{src}, (Col^1_{cbl}, ..., Col^m_{cbl}))$

Notations :

- $\mathcal{T}_{src}$: le nom de la table source
- $\mathcal{T}_{cbl}$: le nom de la table cible
- n : le nombre de colonnes de $\mathcal{T}_{src}$
- $C^i_{\mathcal{T}_{src}}$: le nom de la colonne i de la table $\mathcal{T}_{src}$ où $1 \leq i \leq n$
- Col_{src} : le nom de la colonne qui a été scindée $\wedge$ $Col_{src} \in \{C^1_{\mathcal{T}_{src}}, ..., C^n_{\mathcal{T}_{src}}\}$
- m : le nombre de colonnes en lequel la colonne Col_{src} a été scindée
- Col^j_{cbl} : le nom du $j^{\text{ième}}$ composant de la colonne Col_{cbl} où $1 \leq j \leq m$

Préconditions :

- $\mathcal{T}_{src} \neq \mathcal{T}_{cbl}$ (voir remarque page 55)
- La valeur de la colonne Col_{src} est la concaténation des valeurs des colonnes Col^1_{cbl} à Col^m_{cbl}

Alors, la vue permettant d'accéder à la table $\mathcal{T}_{cbl}$ de la même manière que l'on accédait à $\mathcal{T}_{src}$ peut-être créée comme suit :

CREATE VIEW $\mathcal{T}_{src}(\mathcal{C}^1_{\mathcal{T}_{src}}, ..., \mathcal{C}ol_{src}, ..., \mathcal{C}^n_{\mathcal{T}_{src}})$ AS
SELECT $\mathcal{C}^1_{\mathcal{T}_{cbl}}, ..., concat(\mathcal{C}ol^1_{cbl}, ..., \mathcal{C}ol^m_{cbl}), ..., \mathcal{C}^{n+m-1}_{\mathcal{T}_{cbl}}$
FROM $\mathcal{T}_{cbl}$

où *concat()* est la fonction de concaténation du SGBD

5.4 Scinder une table

Scinder la table $\mathcal{T}_{src}$ en deux tables, $\mathcal{T}_{cbl}$ et $\mathcal{T}_{nv}$

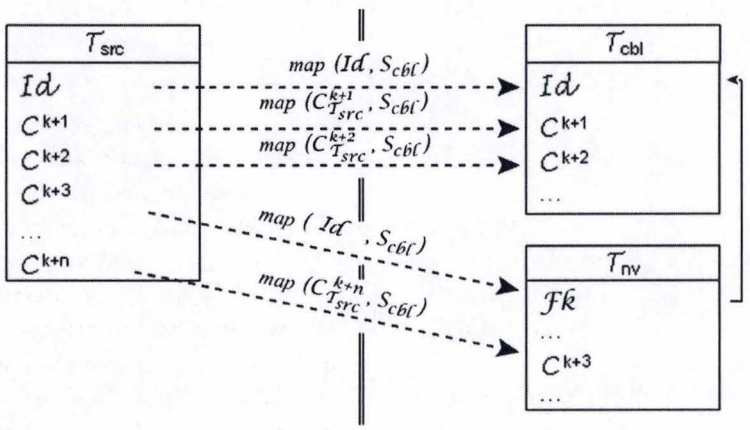


FIGURE 5.4 – *Scinder une table*

Signature de la transformation : $split\_table(\mathcal{T}_{src}, \mathcal{T}_{nv}, \{\mathcal{C}ol_{cbl}\}, \{\mathcal{C}ol_{nv}\})$

Notations :

- $\mathcal{S}_{src}$: le schéma source
- $\mathcal{S}_{cbl}$: le schéma cible
- $\mathcal{T}_{src}$: le nom de la table source
- $\mathcal{T}_{cbl}$: le nom de la table cible dans $\mathcal{S}_{cbl} \equiv \mathcal{T}_{src}$ dans $\mathcal{S}_{src}$
- $\mathcal{T}_{nv}$: le nom de la nouvelle table de $\mathcal{S}_{cbl}$ résultant de la transformation de $\mathcal{T}_{src}$
- n : le nombre de colonnes de $\mathcal{T}_{src}$ hors identifiant primaire
- $\mathcal{I}d_{\mathcal{T}_{src}}$: l'identifiant primaire de $\mathcal{T}_{src}$
- $\mathcal{I}d_{\mathcal{T}_{cbl}}$ est l'identifiant primaire de la table $\mathcal{T}_{cbl}$
- $\mathcal{F}k_{\mathcal{T}_{nv}}$ est la clé étrangère de la table $\mathcal{T}_{nv}$ référençant la table $\mathcal{T}_{cbl}$
- k : le nombre de colonnes constituant $\mathcal{I}d_{\mathcal{T}_{src}}$

- $C^i_{\mathcal{T}_{src}}$: le nom de la colonne i de la table $\mathcal{T}_{src}$ où $1 \leq i \leq k+n$
- $\{Col_{cbl}\}$, $\{Col_{nv}\}$: la répartition des colonnes de $\mathcal{T}_{src}$ dans chacune des deux tables

Soit une fonction de mapping qui associe à deux versions du même schéma un ensemble de couples de colonnes tel que le deuxième élément du couple est la transformation du premier élément :

$$mapping(\mathcal{S}_{src}, \mathcal{S}_{cbl}) \rightarrow \{(\mathcal{C}_{src}, \mathcal{C}_{cbl}) : map(\mathcal{C}_{src}, \mathcal{S}_{cbl}) = \mathcal{C}_{cbl} \wedge \mathcal{C}_{src} \in \mathcal{S}_{src} \wedge \mathcal{C}_{cbl} \in \mathcal{S}_{cbl}\}$$

$$map(\mathcal{C}_{src}, \mathcal{S}_{cbl}) \rightarrow \{\mathcal{C}_{cbl} \in \mathcal{S}_{cbl} : (\mathcal{C}_{src}, \mathcal{C}_{cbl}) \in mapping(\mathcal{S}_{src}, \mathcal{S}_{cbl}) \wedge \mathcal{C}_{src} \in \mathcal{S}_{src}\}$$

Précondition :

- $\mathcal{T}_{src} \neq \mathcal{T}_{cbl} \neq \mathcal{T}_{nv}$ (voir remarque page 55)

Alors, la vue permettant d'accéder à la table $\mathcal{T}_{cbl}$ de la même manière que l'on accédait à $\mathcal{T}_{src}$ peut-être créée comme suit [1] :

CREATE VIEW $\mathcal{T}_{src}(\mathcal{I}d_{\mathcal{T}_{src}}, C^{k+1}_{\mathcal{T}_{src}}, ..., C^{k+n}_{\mathcal{T}_{src}})$ AS
SELECT $map(\mathcal{I}d_{\mathcal{T}_{src}}, \mathcal{S}_{cbl}), map(C^{k+1}_{\mathcal{T}_{src}}, \mathcal{S}_{cbl}), ..., map(C^{k+n}_{\mathcal{T}_{src}}, \mathcal{S}_{cbl})$
FROM $\mathcal{T}_{cbl}, \mathcal{T}_{nv}$
WHERE $\mathcal{F}k_{\mathcal{T}_{nv}} = \mathcal{I}d_{\mathcal{T}_{cbl}}$

5.5 Fusionner deux tables

Fusionner deux tables, $\mathcal{T}_{src_1}$ et $\mathcal{T}_{src_2}$, en une table $\mathcal{T}_{cbl}$

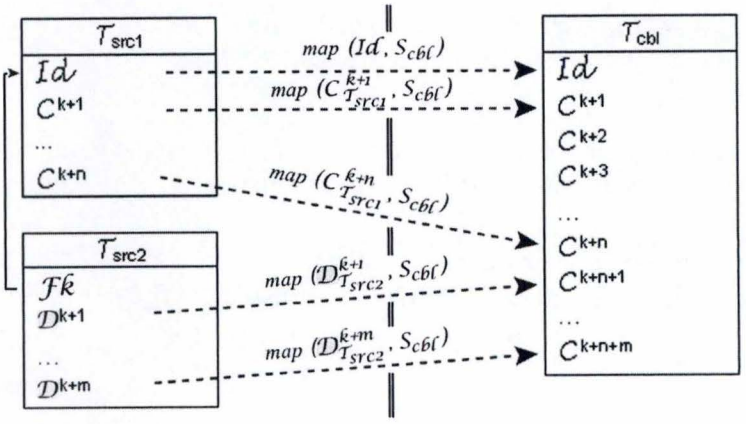


FIGURE 5.5 – *Fusionner deux tables*

Signature de la transformation : $merge\_tables(\mathcal{T}_{src_1}, \mathcal{T}_{src_2})$

1. Par souci de lisibilité, nous n'allons pas détailler le "mapping" de chacune des k colonnes constituant l'identifiant $\mathcal{I}d_{\mathcal{T}_{src}}$. Par abus de langage, nous le noterons simplement $map(\mathcal{I}d_{\mathcal{T}_{src}}, \mathcal{S}_{cbl})$

Notations :

- $\mathcal{S}_{src}$: le schéma source
- $\mathcal{S}_{cbl}$: le schéma cible
- $\mathcal{T}_{src_1}$: le nom de la première table source
- $\mathcal{T}_{src_2}$: le nom de la deuxième table source
- $\mathcal{T}_{cbl}$: le nom de la table cible $\equiv$ table résultant de la fusion de $\mathcal{T}_{src_1}$ et $\mathcal{T}_{src_2}$
- n : le nombre de colonnes de $\mathcal{T}_{src_1}$ hors identifiant primaire
- m : le nombre de colonnes de $\mathcal{T}_{src_2}$ hors clé étrangère
- $\mathcal{I}d_{\mathcal{T}_{src_1}}$: l'identifiant primaire de $\mathcal{T}_{src_1}$
- $\mathcal{I}d_{\mathcal{T}_{cbl}}$: l'identifiant primaire de $\mathcal{T}_{cbl}$
- k : le nombre de colonnes constituant $\mathcal{I}d_{\mathcal{T}_{src_1}}$
- $\mathcal{C}^i_{\mathcal{T}_{src_1}}$: le nom de la colonne i de la table $\mathcal{T}_{src_1}$ où $1 \leq i \leq k+n$
- $\mathcal{F}k_{\mathcal{T}_{src_2}}$ est la clé étrangère de la table $\mathcal{T}_{src_2}$ référençant la table $\mathcal{T}_{src_1}$
- $\mathcal{D}^j_{\mathcal{T}_{src_2}}$: le nom de la colonne j de la table $\mathcal{T}_{src_2}$ où $1 \leq j \leq k+m$

Soit une fonction de mapping qui associe à deux versions du même schéma un ensemble de couples de colonnes tel que le deuxième élément du couple est la transformation du premier élément :

$$mapping(\mathcal{S}_{src}, \mathcal{S}_{cbl}) \rightarrow \{(\mathcal{C}_{src}, \mathcal{C}_{cbl}) : map(\mathcal{C}_{src}, \mathcal{S}_{cbl}) = \mathcal{C}_{cbl} \wedge \mathcal{C}_{src} \in \mathcal{S}_{src} \wedge \mathcal{C}_{cbl} \in \mathcal{S}_{cbl}\}$$

$$map(\mathcal{C}_{src}, \mathcal{S}_{cbl}) \rightarrow \{\mathcal{C}_{cbl} \in \mathcal{S}_{cbl} : (\mathcal{C}_{src}, \mathcal{C}_{cbl}) \in mapping(\mathcal{S}_{src}, \mathcal{S}_{cbl}) \wedge \mathcal{C}_{src} \in \mathcal{S}_{src}\}$$

Préconditions :

- $\mathcal{T}_{src_1} \neq \mathcal{T}_{src_2} \neq \mathcal{T}_{cbl}$ (voir remarque page 55)
- Il existe une relation d'équivalence entre $\mathcal{T}_{src_1}$ et $\mathcal{T}_{src_2}$
- $\mathcal{I}d_{\mathcal{T}_{src_1}} \equiv \mathcal{F}k_{\mathcal{T}_{src_2}}$

Alors, les vues permettant d'accéder à la table $\mathcal{T}_{cbl}$ de la même manière que l'on accédait à $\mathcal{T}_{src_1}$ et $\mathcal{T}_{src_2}$ peuvent-être créées comme suit [2] :

CREATE VIEW $\mathcal{T}_{src_1}(\mathcal{I}d_{\mathcal{T}_{src_1}}, \mathcal{C}^{k+1}_{\mathcal{T}_{src_1}}, ..., \mathcal{C}^{k+n}_{\mathcal{T}_{src_1}})$ AS
SELECT $map(\mathcal{I}d_{\mathcal{T}_{src_1}}, \mathcal{S}_{cbl}), map(\mathcal{C}^{k+1}_{\mathcal{T}_{src_1}}, \mathcal{S}_{cbl}), ..., map(\mathcal{C}^{k+n}_{\mathcal{T}_{src_1}}, \mathcal{S}_{cbl})$
FROM $\mathcal{T}_{cbl}$

CREATE VIEW $\mathcal{T}_{src_2}(\mathcal{F}k_{\mathcal{T}_{src_2}}, \mathcal{D}^{k+1}_{\mathcal{T}_{src_2}}, ..., \mathcal{D}^{k+m}_{\mathcal{T}_{src_2}})$ AS
SELECT $map(\mathcal{F}k_{\mathcal{T}_{src_2}}, \mathcal{S}_{cbl}), map(\mathcal{D}^{k+1}_{\mathcal{T}_{src_2}}, \mathcal{S}_{cbl}), ..., map(\mathcal{D}^{k+m}_{\mathcal{T}_{src_2}}, \mathcal{S}_{cbl})$
FROM $\mathcal{T}_{cbl}$

2. Par souci de lisibilité, nous n'allons pas détailler le "mapping" de chacune des k colonnes constituant l'identifiant $\mathcal{I}d_{\mathcal{T}_{src_1}}$ et la clé $\mathcal{F}k_{\mathcal{T}_{src_2}}$. Par abus de langage, nous les noterons respectivement $map(\mathcal{I}d_{\mathcal{T}_{src_1}}, \mathcal{S}_{cbl})$ et $map(\mathcal{F}k_{\mathcal{T}_{src_2}}, \mathcal{S}_{cbl})$

5.6 Limitations

L'impact négatif de ces transformations sur le code applicatif existant n'est pas toujours totalement pris en charge par la création de la vue, ceci va dépendre de la nature de la requête. Le tableau ci-dessous présente un récapitulatif des limites de l'utilisation des vues pour l'évolution :

	SELECT	INSERT	UPDATE	DELETE
Renommer une table	✓	✓	✓	✓
Renommer une colonne	✓	✓	✓	✓
Scinder une colonne	✓	X	X	✓
Scinder une table	✓	X	X	X
Fusionner deux tables	✓	X	X	✓

Ces limitations sont dûes aux restrictions quant à la possibilité de mettre à jour une vue (voir Chapitre 1 Section 1.5).

Chapitre 6

Outils

Maintenant que nous avons défini une méthodologie (Chapitre 4) et que nous avons formalisé les règles de génération des vues (Chapitre 5), nous pouvons automatiser le processus. Pour ce faire nous allons développer un prototype au sein de l'outil de modélisation DB-Main. Nous commencerons donc par présenter DB-Main (Section 6.1). Ensuite, nous présentons le prototype en tant que tel (Section 6.2), la possibilité de tester la/les vue(s) générée(s) (Sous-section 6.2.4), et nous illustrerons tout cela par un exemple (Sous-section 6.2.5). Enfin, nous clôturerons le chapitre en détaillant les signatures des fonctions permettant la génération des vues (Sous-section 6.2.6).

6.1 DB-Main

6.1.1 Présentation

DB-Main [dbm] est un outil de modélisation pour les applications de base de données. Créé au début des années 90 dans le cadre d'un projet de recherche du Laboratoire d'Ingéniérie des applications de Bases de Données des FUNDP, il est maintenant développé et distribué par REVER s.a.

Actuellement, la dernière version stable de DB-Main est la version 9.0. C'est cette version que nous utiliserons pour les besoins de ce mémoire.

DB-Main manipule des fichiers *.lun* contenant des projets dans lesquels on peut créer un ou plusieurs schémas. La création d'un nouveau schéma peut se faire depuis la copie d'un schéma existant. Au sein d'un projet, un schéma sera identifié par son nom et son numéro de version, notation : `<NOM>/<Version>`, exemple : `SCHEMA_SOURCE/1-1`

6.1.2 Transformation

DB-Main propose, entre autres, des outils d'aide à la transformation de schéma.
Ils sont regroupés sous l'onglet "Transform" (Figure 6.1).

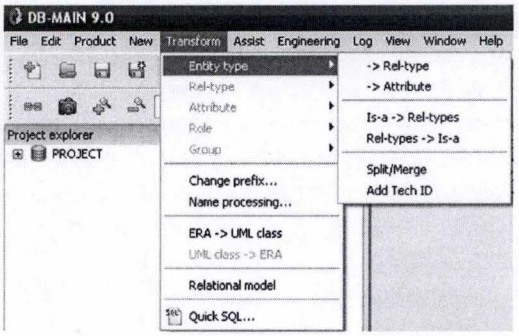


FIGURE 6.1 – *DB-Main 9.0 - onglet "Transform"*

6.1.3 Mapping

DB-Main associe des Méta-propriétés à ses Méta-objets (projet, schéma, type d'entité, attribut,...). Une de ces Méta-propriétés porte le nom de "*MappingOID*", elle est associée par défaut à certains Méta-objets [1] et sert au mécanisme de mapping. En effet, lors de l'initialisation du mapping, le "*MappingOID*" de chacun des Méta-objets se voit associer un numéro unique au sein du schéma. Lors de la transformation d'un schéma, le où les Méta-objets impliqués dans la transformation conserveront leur "*MappingOID*" de sorte que, si une copie du schéma avait été effectuée au préalable, il est possible de retracer l'historique des transformations.

6.1.3.1 Exemples

Renommer une table

Supposons que l'on renomme la table `CLIENT` en `CUSTOMER`.
Sur la Figure 6.2 on voit que le "*MappingOID*" reste le même dans chacune de deux versions du schéma.

SCHEMA_SOURCE/1

CLIENT	*MappingOID*
	1234
NCli	*151*
Nom	*156*
Adresse	*74*
Localite	*158*
Cat	*78*
Compte	*80*

SCHEMA_CIBLE/1

CUSTOMER	*MappingOID*
	1234
NCli	*151*
Nom	*156*
Adresse	*74*
Localite	*158*
Cat	*78*
Compte	*80*

FIGURE 6.2 – *"MappingOID" - Renommer une table*

1. La Méta-propriété "*MappingOID*" est associée par défaut aux Méta-objets suivants : Atomic attribute, Collection, Compound attribute, Entity type, Group, Processing unit, Rel-type, Role.

Scinder une colonne

Supposons que l'on scinde la colonne `Adresse` en deux colonnes `Num` et `Rue`.
Sur la Figure 6.3 on voit que le "*MappingOID*" de chacune des colonnes `Num` et `Rue` est celui de la colonne `Adresse` du schéma source.

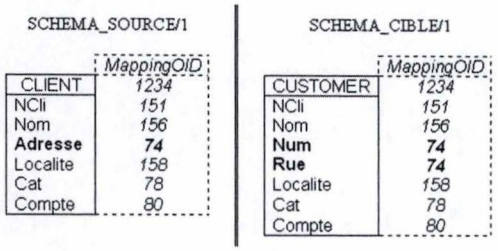


FIGURE 6.3 – *"MappingOID" - Scinder une colonne*

6.1.4 Plug-in

Via la JIDBM (Java Interface for DB-Main), DB-Main fournit un ensemble de classes permettant aux utilisateurs d'écrire leurs propres plug-in pour accéder en lecture et en écriture à leurs projets selon leurs besoins.

Il y a deux possibilités pour exploiter les plug-in développés par l'utilisateur. Soit en les lançant via DB-Main à l'aide du bouton "Execute plug-in", soit en lançant le fichier *.class* en dehors de DB-Main.

Illustrons l'utilisation de cette API par un petit bout de code qui affiche le nom des tables et le nom des colonnes du schéma :

```
import jidbm.*;
...
   DBMProject sys = new DBMProject();
   if (sys != null) {
      // Chargement du schéma
      DBMSchema sch = sys.getFirstProductSchema();
      if (sch != null) {
         // Chargement de la première table du schéma
         DBMEntityType table = sch.getFirstDataObjectEntityType();
         while (table != null){
            System.out.println(table.getName());
            // Chargement de la première colonne de la table
            DBMAttribute colonne = table.getFirstAttribute();
            while(colonne!=null){
```

```
                System.out.println("  "+colonne.getName());
                colonne = table.getNextAttribute(colonne);
            }
            table = sch.getNextDataObjectEntityType(table);
        }
    }
}
...
```

Le générateur de vues décrit à la section 6.2 a été écrit en utilisant cette API Java.

6.2 Générateur de vues

6.2.1 Présentation

En se basant sur la méthodologie exposée au Chapitre 4 et sur les règles de génération de vues définies au Chapitre 5, à l'aide de la JIDBM (voir sous-section 6.1.4), nous avons développé un plug-in DB-Main. Nous appellerons ce plug-in "*Générateur de vues*".

Notre approche consiste à demander le minimum d'informations [2] à l'utilisateur, nous souhaitons juste savoir ce qu'il a fait. A partir de ces quelques informations, notre *Générateur de vues* exploite le mapping pour compléter la signature de la transformation, et produire la/les vues nécessaires à la compatibilité du code applicatif existant.

$$INPUTS \Rightarrow MAPPING \Rightarrow SIGNATURE \Rightarrow VUE(S)$$

6.2.2 Fonctionnement général

Le fonctionnement général du prototype est illustré à la Figure 6.4.

Nous pouvons le décrire en 3 étapes :

Tout d'abord, l'utilisateur interagit avec l'interface du *Générateur de vues*, il fait ses choix parmi les options qui lui sont proposées. En fonction de ses choix, l'interface est rafraîchie grâce aux données issues du fichier DB-Main.

Ensuite, une fois ses choix effectués, il appuye sur le bouton "*Générer vue(s)*", les options choisies sont alors envoyées au processus de génération de vues qui exploite DB-Main et le principe de mapping pour pouvoir envoyer à l'interface la signature complète de la transformation ainsi que la vue générée.

2. L'information qui nous est nécessaire est la suivante : le schéma de base de données avant transformation, le schéma après transformation, la nature de la transformation, et le(s) objet(s) du schéma concerné(s)

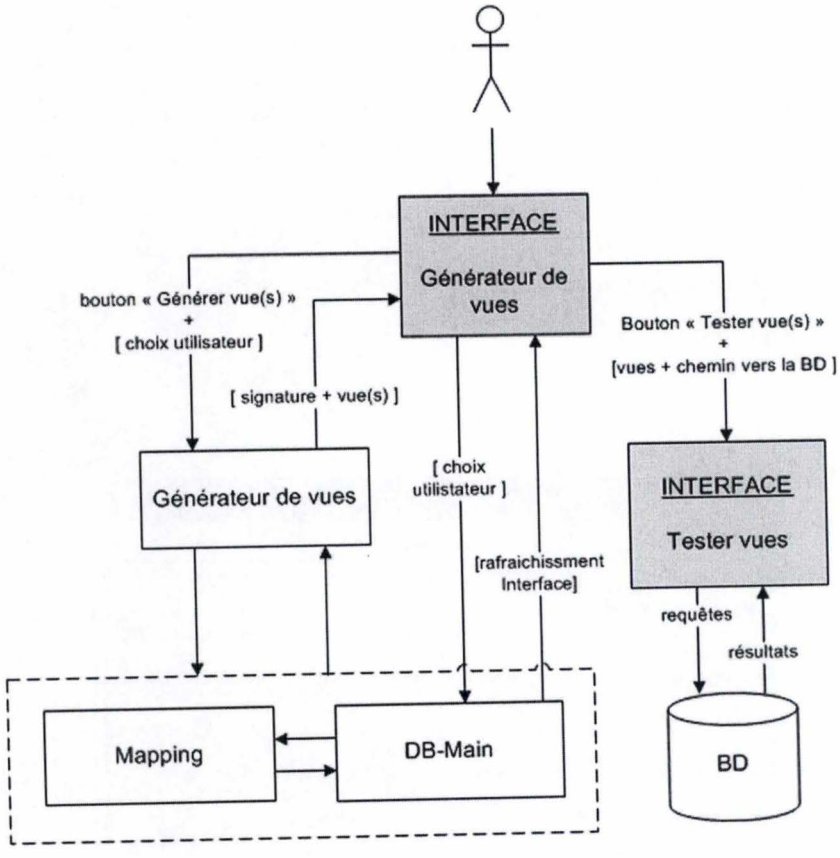


FIGURE 6.4 – *Fonctionnement général du "Générateur de vues"*

Enfin, en appuyant sur le bouton "*tester vue(s)*", l'utilisateur peut tester visuellement l'efficacité de la vue générée. Pour ce faire, il peut soumettre des requêtes SQL et comparer le résultat obtenu après transformation avec/sans vue(s) au résultat obtenu avant transformation.

6.2.3 Utilisation

Pour pouvoir utiliser le *Générateur de vues*, au préalable, il faut avoir :

- créé un projet DB-Main et l'avoir sauvé dans un fichier *.lun*
- créé un schéma dans ce projet
- initialisé le mapping sur le schéma créé
- créé une copie du schéma original et y appliquer la transformation

Il y a deux manières d'utiliser le plug-in *Générateur de vues* :

- au sein de DB-Main via le bouton "Execute plug-in".
- en dehors, directement via le fichier `EvolutionManager.class`

Si on utilise la deuxième méthode, au lancement du plug-in, il nous sera demandé de choisir le fichier DB-Main contenant le projet à traiter.

La fenêtre principale du *Générateur de vues* est divisée en quatres parties (cfr. Figure 6.5) :

1. Bases de données
2. Transformations
3. Signature de la transforamtion
4. Vue(s) générée(s)

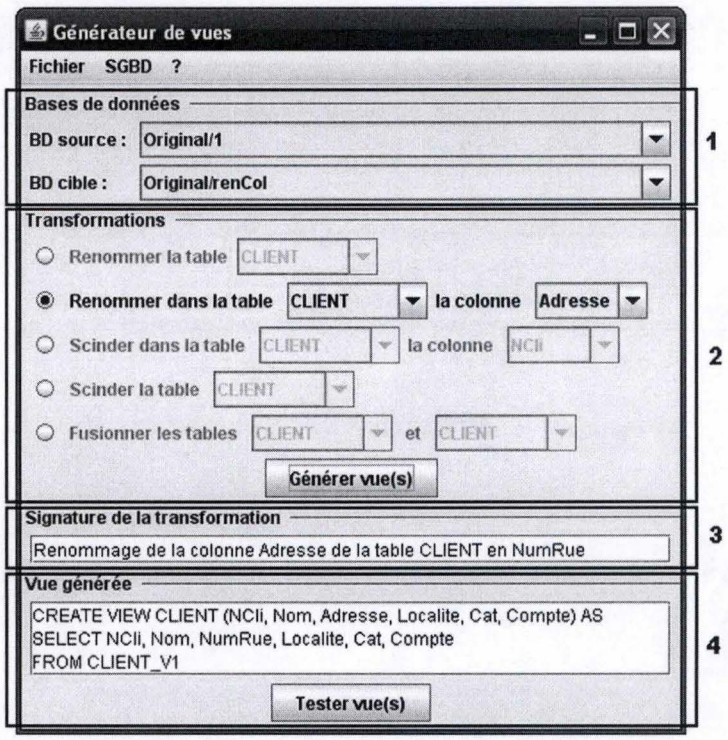


FIGURE 6.5 – *Interface du "Générateur de vues"*

Après s'être assuré que le SGBD sélectionné sous l'onglet *SGBD* est bien celui utilisé, nous pouvons préciser les paramètres d'entrée dans les différentes parties de l'interface :

1. Bases de données
Dans cette partie, il nous est demandé d'indiquer la *BD source*, schéma de base de données avant transformation, et la *BD cible*, schéma de base de données après transformation. Pour ce faire nous pouvons choisir parmi les différents schémas du projet DB-Main.

2. Transformations
Ensuite, il nous est demandé de choisir la nature de la transformation qui a été réalisée. Pour chacune des transformations proposées, il nous sera demandé de préciser certaines informations de manière à rendre possible la génération de la vue.

Une fois ce choix effectué, nous pouvons appuyer sur le bouton "*Générer vue(s)*" pour produire la/les vue(s).

3. Signature de la transformations
Lorsque les choix précédents ont été effectués et que l'on a appuyé sur le bouton "*Générer vue(s)*", le champs de texte de la partie "Signature de la transformation" contient la signature complète de la transformation. Ce champ de texte sert aussi à nous signaler une erreur en cas de problème lors de la génération de la/des vue(s).

4. Vue(s) générée(s)
C'est dans cette partie qu'est affichée la/les vue(s) générée(s).
Au bas de cette partie, le bouton "*tester vue(s)*" nous envoie vers une fenêtre qui nous permet de tester l'efficacité de la/des vue(s) générée(s) (voir section 6.2.4).

6.2.4 Tester vue(s)

Une fois la/les vue(s) générée(s), nous pouvons la/les tester en appuyant sur le bouton "*tester vue(s)*".

Ce module s'inspire du code développé par David Flanagan disponible à l'adresse internet suivante :

`http ://www.oreillynet.com/pub/a/oreilly/java/news/javaex_1000.html`

Pour pouvoir utiliser ce module de test, au préalable, il faut avoir :

- démarré le SGBD
- créé les bases données correspondantes aux schémas DB-Main dans le SGBD de notre choix
- défini une "*Meta-propriété*" pour chacun des schémas DB-Main utilisé. Ceci se fait via l'option "*Meta-properties*" de l'onglet "*Product*". Le nom de la meta-propriété doit être de la forme : "*connectionPathSGBD*" où SGDB est "*MYSQL*" ou "*ORACLE*". La valeur de la meta-propriété doit être le chemin vers la base de données correspondante au schéma DB-Main, par exemple : `jdbc :mysql//127.0.0.1 :3306/trans01_source`

La fenêtre principale est divisée en trois parties (cfr. Figure 6.6) :

1. Requête(s)
2. BD Source
3. BD Cible

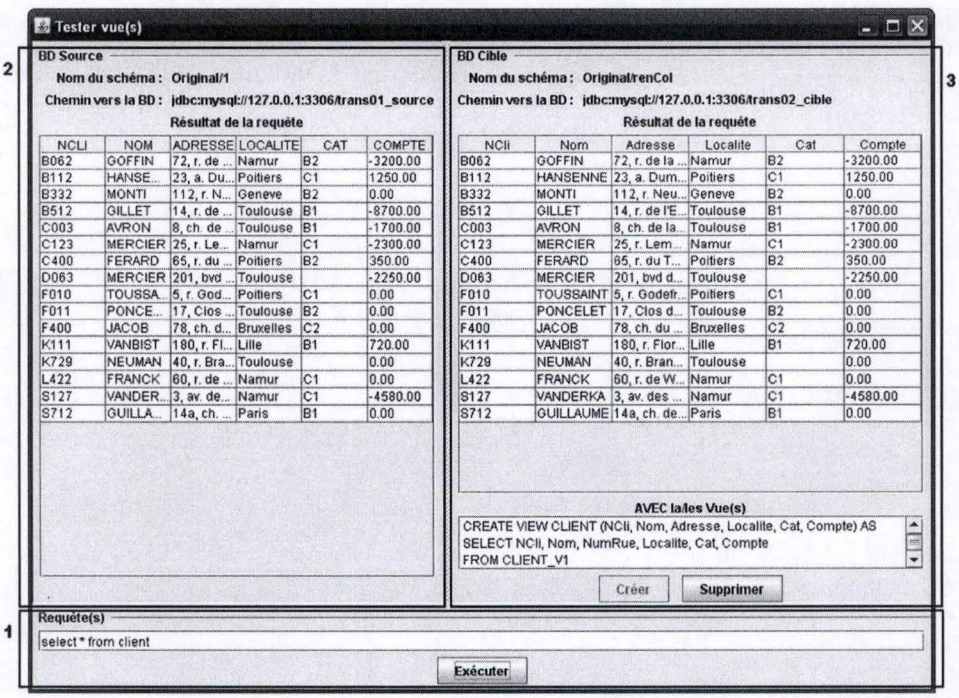


FIGURE 6.6 – *Interface de "Tester vue(s)"*

Détaillons les différentes parties de l'interface :

1. Requête(s)
Dans le but de tester l'efficacité de la vue, cette partie nous permet d'effectuer une requête sur les deux bases de données pour ensuite comparer les résulats obtenus.

2. BD Source
Cette partie présente le résultat obtenu en executant une requête sur la base de données source, base de données avant transformation.

3. BD Cible
Cette partie présente le résultat obtenu en executant une requête sur la base de données cible, base de données après transformation. En bas de cette partie, est affichée la vue générée précédemment, nous la créons en appuyant sur le bouton "Créer" ou la supprimons en appuyant sur le bouton "Supprimer".

6.2.5 Exemple d'utilisation illustré

Après avoir présenté le "*Générateur de vues*", nous allons illustrer son utilisation par un exemple. Pour ce faire, prenons le cas du scindage d'une table `CLIENT` en deux tables, `CLIENT` et `ADRESSE`. La table `ADRESSE` étant la transformation en une nouvelle table de la colonne `Adresse` de la table `CLIENT`.

La Figure 6.7 présente les schémas relationnels logiques avant (schéma source) et après (schéma cible) transformation.

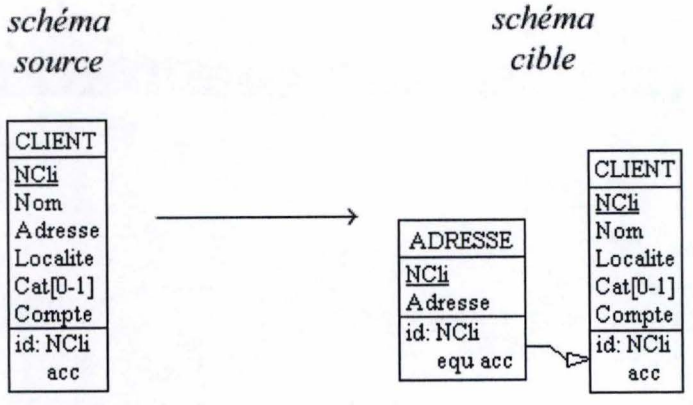


FIGURE 6.7 – *Schéma source et schéma cible*

Il est impératif d'initialiser le mapping sur le schéma source avant de le copier dans le but de le transformer. En effet, le schéma cible doit être issu d'une copie du schéma source à laquelle on a appliqué la transformation, et pour répondre aux préconditions d'utilisation du "*Générateur de vues*", il faut qu'il existe un lien de mapping entre les deux schémas. L'initialisation du mapping peut se faire via la classe `InitMappingOID` développée par Anthony Cleve. La copie du schéma est elle réalisée au sein de DB-Main à l'aide de l'option ***Copy product...*** de l'onglet ***Product*** (Figure 6.8).

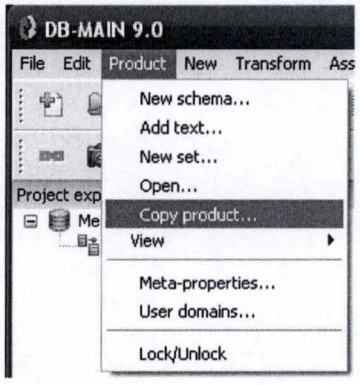


FIGURE 6.8 – *Onglet* ***Copy product...*** *de DB-Main*

Il y a deux manières de lancer le "*Générateur de vues*", la première consiste à utiliser le bouton "Excecute plug-in" de l'interface DB-Main, la seconde à exécuter la classe java `EvolutionManager`. Lorsque l'on opte pour la deuxième solution, il nous est demandé de choisir le fichier *.lun*[3] contenant les schémas à analyser (Figure 6.9).

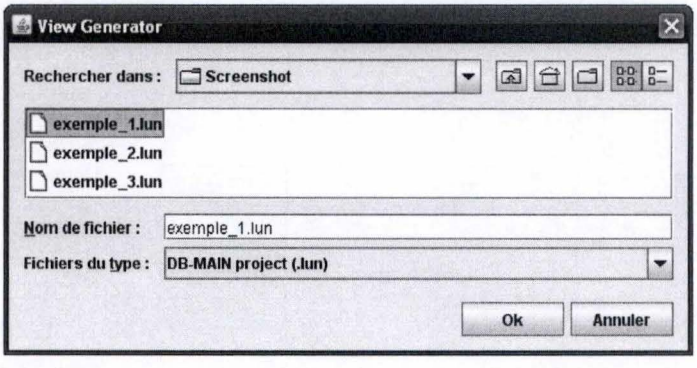


FIGURE 6.9 – *Sélecteur de fichier du "Générateur de vues"*

3. extention des fichiers DB-Main

Le "*Générateur de vues*" est maintenant lancé (Figure 6.10).

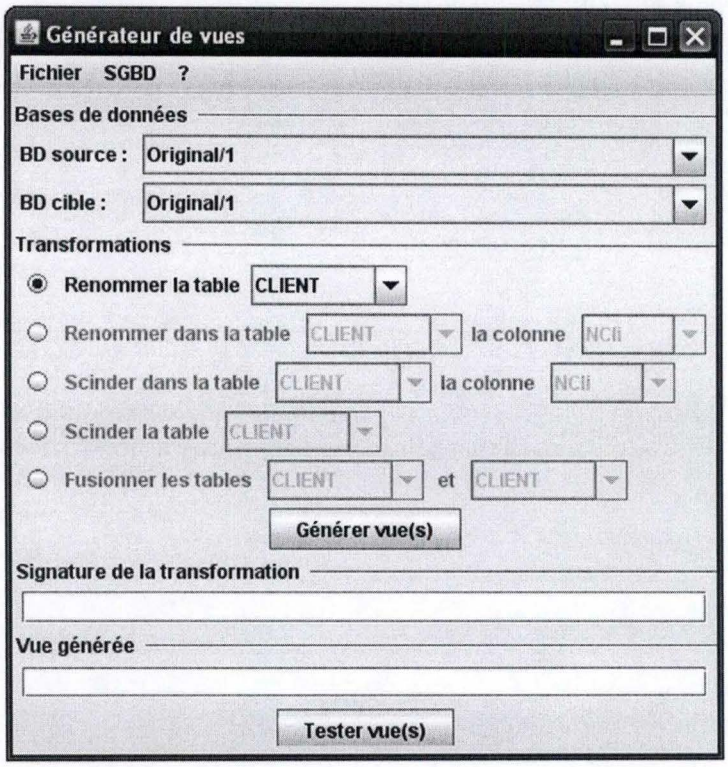


FIGURE 6.10 – *Le "Générateur de vues"*

Pour des raisons de syntaxe propres aux SGBD, notamment au niveau de la fonction de concaténation, il est préferable de vérifier que le SGBD que nous utilisons est bien celui sélectionné sous l'onglet ***SGBD*** (Figure 6.11).

FIGURE 6.11 – *Choix du SGBD*

Il faut maintenant préciser le schéma source et le schéma cible. Les intitulés des schémas présentés dans les listes déroulantes sont l'ensemble des schémas présents au sein du projet du fichier *.lun* traité (Figure 6.12).

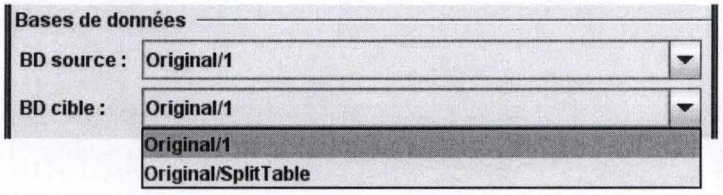


FIGURE 6.12 – *Choix des schémas source et cible*

Une fois les schémas sélectionnés, il est demandé de préciser ce qui a été fait comme transformation. Dans notre exemple, nous indiquerons juste avoir scindé la table `CLIENT` (Figure 6.13).

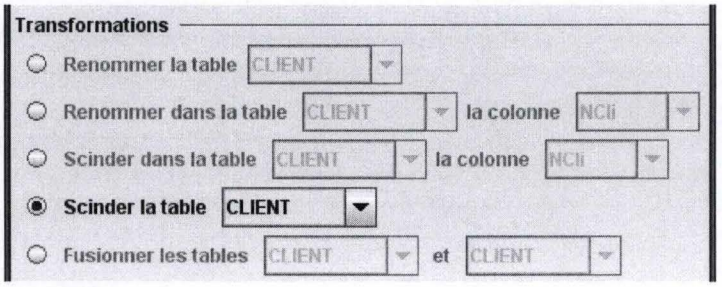


FIGURE 6.13 – *Choix de la transformation*

Les informations nécessaires étant collectées, nous pouvons maintenant appuyer sur le bouton "*Générer vue(s)*" (Figure 6.14). Ceci a pour effet de lancer le processus qui consiste dans un premier temps à exploiter le mapping pour compléter la signature de la transformation et ensuite générer la/les vue(s). Le champ de texte de la partie "Signature de la transformation" contient la signature complète de la transformation et le champ "Vue Générée" contient la/les vue(s) (Figure 6.15).

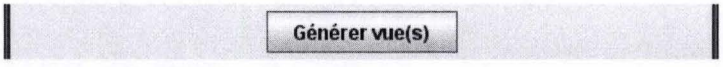


FIGURE 6.14 – *Le bouton "Générer vue(s)"*

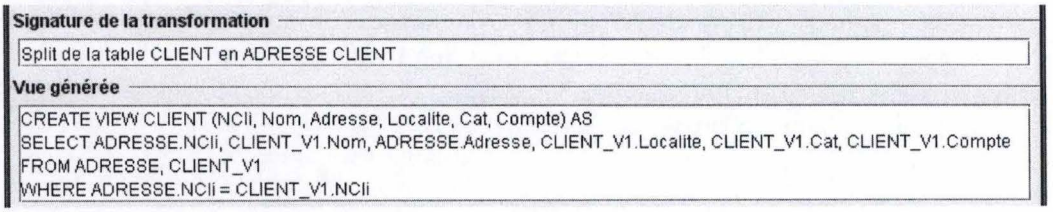


FIGURE 6.15 – *Signature de la transformation et vue associée*

Si nous le souhaitons, nous pouvons tester la vue générée à l'aide du bouton "*tester vue(s)*" (Figure 6.16).

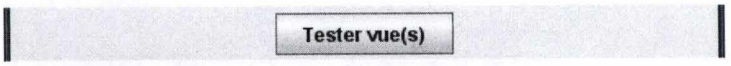


FIGURE 6.16 – *Le bouton "tester vue(s)"*

Pour ce faire, au préalable, pour répondre aux préconditions d'utilisation, il faut :

- produire le code SQL de création des bases de données compatible avec notre SGBD (dans notre cas, MySQL). La génération du code SQL MySQL se fait via l'option ***Generate > MySQL...*** de l'onglet ***File*** de DB-Main.
- créer les deux bases de données source et cible :
  `mysql> CREATE DATABASE sch_source ; CREATE DATABASE sch_cible ;`
- via l'option ***Meta-properties...*** de l'onglet ***Product***, créer une "*Méta-propriété*" attachée au "*Méta-objets*" de type "*Schéma*" nommée `connectionPathMySQL`. Nous l'initialisons à *jdbc :mysql//127.0.0.1 :3306/sch_source* pour le schéma source et à *jdbc :mysql//127.0.0.1 :3306/sch_cible* pour le schéma cible (Figure 6.17).

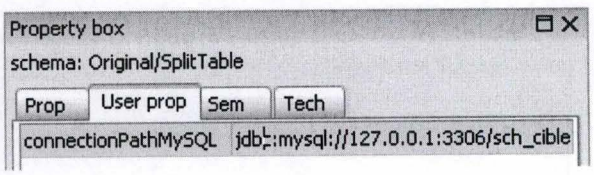


FIGURE 6.17 – *DB-Main - User Prop*

Les préconditions étant maintenant remplies, nous pouvons tester la vue en soumettant une requête au SGBD. Nous choisissons de soumettre la requête suivante :
`select * from CLIENT order by NCli`

Lors de la première utilisation, la vue n'est pas créée, elle nous est juste présentée visuellement (Figure 6.18). Si nous soumettons une requête invoquant la vue, celle-ci échouera sur la BD cible (Figure 6.19). Rappelons que la table `CLIENT` a été renommée dans le schéma cible pour laisser place à la vue.

FIGURE 6.18 – *Présentation de la vue*

FIGURE 6.19 – *Erreur - La table `CLIENT` n'existe pas dans la BD cible*

Si par contre, nous modifions notre requête comme ci-dessous de manière à prendre en compte la transformation, la requête n'échouera plus sur la BD cible, par contre elle échouera sur la BD source... (Figure 6.20)

```
select *
from CLIENT_V1, ADRESSE
where CLIENT_V1.NCli=ADRESSE.NCli
order by NCli
```

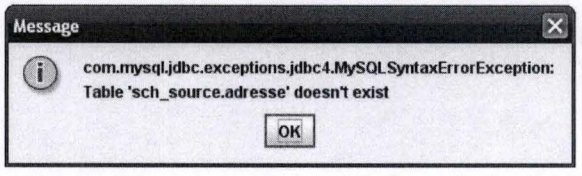


FIGURE 6.20 – *Erreur - La table `ADRESSE` n'existe pas dans la BD source*

Lorsque nous ajoutons la vue à l'aide du bouton "*Créer*", on voit que la requête qui fonctionnait sur la BD source fonctionne maintenant aussi sur la BD cible, un rapide coup d'oeil sur les résultats obtenus nous montre qu'ils sont identiques (Figure 6.21).

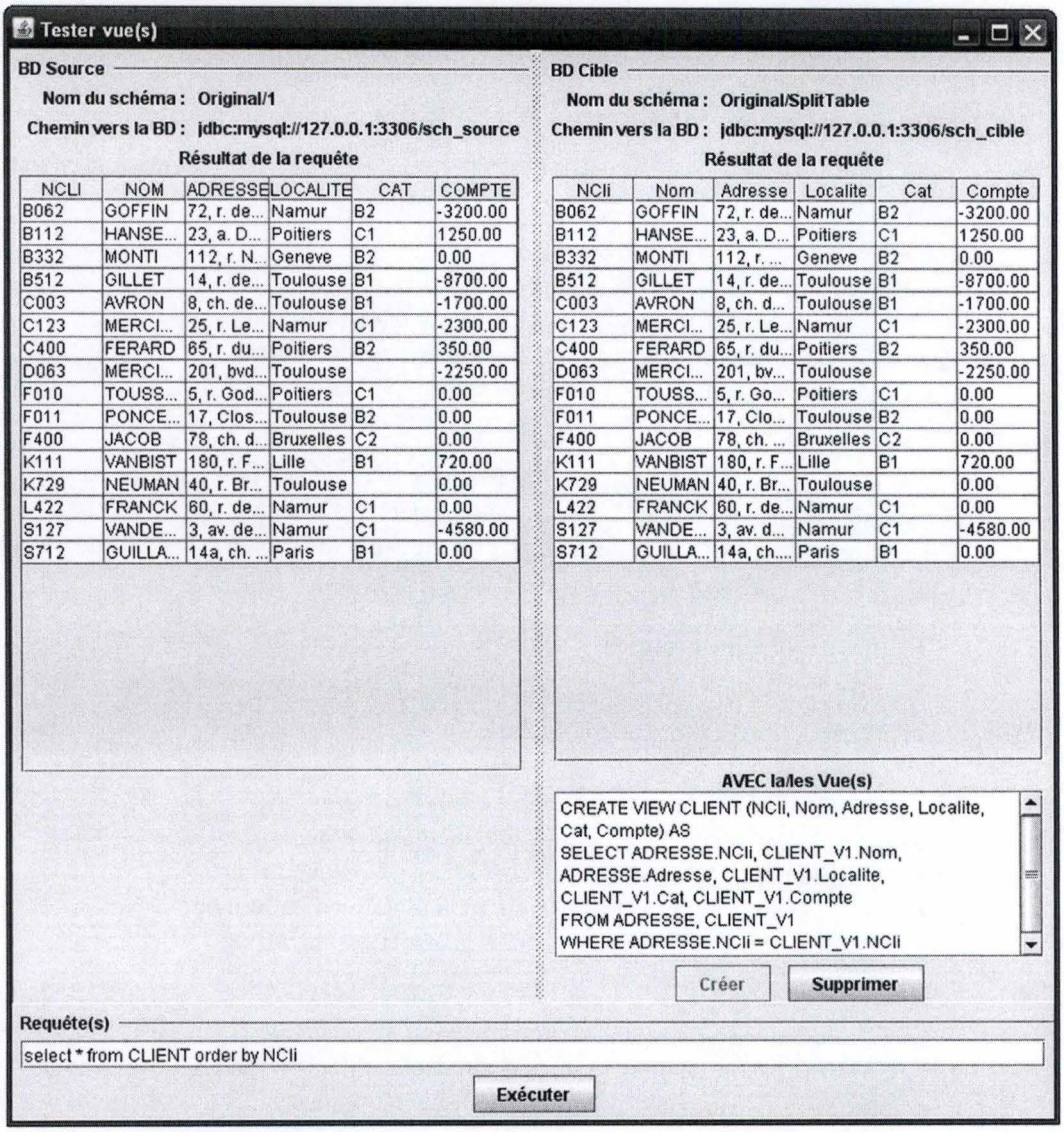


FIGURE 6.21 – *Comparaison visuelle des résultats obtenus*

6.2.6 Signature des fonctions de transformation

Le *Générateur de vues* permet la création de vues pour chacune des transformations présentées au Chapitre 5. Pour ce faire, à chacune des transformations correspond une fonction dans la classe `ViewGeneratorFromMapping.class`.

Détaillons la signature de ces fonctions...

6.2.6.1 Renommer une table

```
public String[] mapping_rename_table(DBMSchema sch_src, DBMSchema sch_cbl,
String tableOldName)
```

Entrées :

- sch_src : le schéma de base de données avant transformation
- sch_cbl : le schéma de base de données après transformation
- tableOldName : le nom de la table avant transformation

Sortie :

- String[ ] : un tableau de String dont le premier élément est la signature complète de la transformation et dont le second élément est la vue générée. En exploitant le mapping, nous pouvons compléter la signature avec le nouveau nom de la table.

6.2.6.2 Renommer une colonne

```
public String[] mapping_rename_column(DBMSchema sch_src, DBMSchema sch_cbl,
String tableName, String columnOldName)
```

Entrées :

- sch_src : le schéma de base de données avant transformation
- sch_cbl : le schéma de base de données après transformation
- tableName : le nom de la table contenant la colonne renommée
- columnOldName : le nom de la colonne avant transformation

Sortie :

- String[ ] : un tableau de String dont le premier élément est la signature complète de la transformation et dont le second élément est la vue générée. En exploitant le mapping, nous pouvons compléter la signature avec le nouveau nom de la colonne.

6.2.6.3 Scinder une colonne

```
public String[] mapping_split_column(DBMSchema sch_src, DBMSchema sch_cbl,
String tableName, String columnName)
```

Entrées :
- sch_src : le schéma de base de données avant transformation
- sch_cbl : le schéma de base de données après transformation
- tableName : le nom de la table contenant la colonne scindée
- columnName : le nom de la colonne scindée

Sortie :
- String[ ] : un tableau de String dont le premier élément est la signature complète de la transformation et dont le second élément est la vue générée. En exploitant le mapping, nous pouvons compléter la signature avec la liste des noms de colonnes en lesquelles la colonne transformée a été scindée.

6.2.6.4 Scinder une table

```
public String[] mapping_split_table(DBMSchema sch_src, DBMSchema sch_cbl,
String tableName)
```

Entrées :
- sch_src : le schéma de base de données avant transformation
- sch_cbl : le schéma de base de données après transformation
- tableName : le nom de la table scindée

Sortie :
- String[ ] : un tableau de String dont le premier élément est la signature complète de la transformation et dont le second élément est la vue générée. En exploitant le mapping, nous pouvons compléter la signature avec le nom de la nouvelle table créée suite au scindage, et connaitre la répartition des colonnes au sein des tables.

6.2.6.5 Fusionner deux tables

```
public String[] mapping_merge_tables(DBMSchema sch_src, DBMSchema sch_cbl,
String tableName1, String tableName2)
```

Entrées :
- sch_src : le schéma de base de données avant transformation
- sch_cbl : le schéma de base de données après transformation
- tableName1 et tableName2 : les noms des deux tables fusionnées

Sortie :
- String[ ] : un tableau de String dont le premier élément est la signature complète de la transformation et dont le second élément est la concaténation des deux vues générées. En exploitant le mapping, nous pouvons compléter la signature avec le nom de la table issue de la fusion.

Chapitre 7

Etude de cas

Ce chapitre présente une étude de cas dans laquelle, après évolution de la base de données, nous allons comparer les résultats obtenus en appliquant le processus décrit au Chapitre 4 aux résultats obtenus en modifiant le code applicatif existant. Ceci dans le but de tester le bon fonctionnement de la solution proposée dans ce mémoire et d'en évaluer l'impact au niveau des performances.

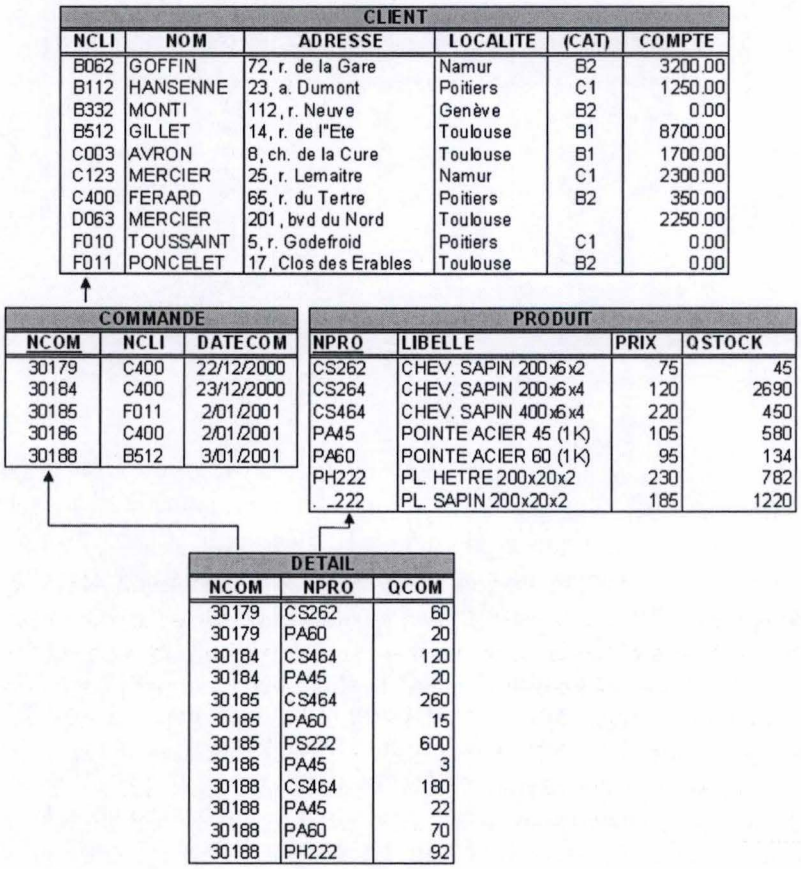
Pour ce faire, nous allons présenter le matériel et les logiciels utilisés (Section 7.1) ainsi que la base de données qui fera l'objet de notre étude (Section 7.2). Après avoir décrit la méthodologie employée (Section 7.3), nous définirons une série de transformations (Section 7.4), et une série de requêtes (Section 7.5) qui nous serviront tout au long de notre étude. Pour chacune des transformations à laquelle nous aurons soumis la série de requêtes, nous réunirons les résultats obtenus dans des tableaux récapitulatifs (Section 7.7) desquels nous discuterons le contenu (Section 7.8).

7.1 Matériel et logiciels

Cette étude de cas a été réalisée sur un ordinateur équipé d'un processeur Intel Core 2 Duo T7100, de 2Gb de RAM, d'un disque dur tournant à 7200 tours/minute, et sur lequel est installé Windows XP SP3. La version de MySQL utilisée est la version 5.1. Notre choix s'est porté sur MySQL pour sa gratuité, sa facilité d'utilisation et surtout pour sa grande popularité dans le développement d'applications en ligne.

7.2 Schéma de la base de données

Notre étude de cas se base sur un exemple repris de [Hai00] illustré à la Figure 7.1.



CLIENT

NCLI	NOM	ADRESSE	LOCALITE	(CAT)	COMPTE
B062	GOFFIN	72, r. de la Gare	Namur	B2	3200.00
B112	HANSENNE	23, a. Dumont	Poitiers	C1	1250.00
B332	MONTI	112, r. Neuve	Genève	B2	0.00
B512	GILLET	14, r. de l'Ete	Toulouse	B1	8700.00
C003	AVRON	8, ch. de la Cure	Toulouse	B1	1700.00
C123	MERCIER	25, r. Lemaitre	Namur	C1	2300.00
C400	FERARD	65, r. du Tertre	Poitiers	B2	350.00
D063	MERCIER	201, bvd du Nord	Toulouse		2250.00
F010	TOUSSAINT	5, r. Godefroid	Poitiers	C1	0.00
F011	PONCELET	17, Clos des Erables	Toulouse	B2	0.00

COMMANDE

NCOM	NCLI	DATECOM
30179	C400	22/12/2000
30184	C400	23/12/2000
30185	F011	2/01/2001
30186	C400	2/01/2001
30188	B512	3/01/2001

PRODUIT

NPRO	LIBELLE	PRIX	QSTOCK
CS262	CHEV. SAPIN 200x6x2	75	45
CS264	CHEV. SAPIN 200x6x4	120	2690
CS464	CHEV. SAPIN 400x6x4	220	450
PA45	POINTE ACIER 45 (1K)	105	580
PA60	POINTE ACIER 60 (1K)	95	134
PH222	PL. HETRE 200x20x2	230	782
PS222	PL. SAPIN 200x20x2	185	1220

DETAIL

NCOM	NPRO	QCOM
30179	CS262	60
30179	PA60	20
30184	CS464	120
30184	PA45	20
30185	CS464	260
30185	PA60	15
30185	PS222	600
30186	PA45	3
30188	CS464	180
30188	PA45	22
30188	PA60	70
30188	PH222	92

FIGURE 7.1 – *Base de données exemple et son contenu [Hai00]*

7.3 Méthodologie

L'objectif de cette étude de cas est d'évaluer l'impact sur les performances au niveau code applicatif suite à l'utilisation de vues pour atténuer les effets négatifs de la transformation d'une base de données. Pour ce faire, nous allons d'abord définir une série de transformations (Section 7.4) et une série de requêtes (Section 7.5).
Ensuite, pour chacune des transformations, nous créerons deux instances de la base de données. La première sera la traduction du schéma avant transformation (BD source), la seconde sera la traduction du schéma après transformation (BD cible).
De manière à s'adapter à la transformation, nous créerons une vue sur la BD cible en utilisant la méthode de génération de vues étudiée dans ce mémoire. Une fois la vue créée, nous interrogerons chacune des bases de données avec la série de requêtes précédemment définie.

De manière à ajouter la dimension "*taille des données*", pour chacune des transformations, nous allons soumettre la série de requêtes à cinq tables, composées respectivement de 10, 100, 1000, 10 000, et 100 000 lignes.

Pour chaque couple transformation/requête (T_X, R_Y), nous rassemblerons les résultats obtenus dans un tableau récapitulatif de la forme :

Transformation$_X$(Type)/Requête$_Y$(Type)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	0 ms	0 ms	0 ms
100	0 ms	0 ms	0 ms
1000	0 ms	0 ms	0 ms
10 000	0 ms	0 ms	0 ms
100 000	0 ms	0 ms	0 ms

où "Tps exec." est le temps d'exécution de la requête exprimé en milliseconde (ms)

7.4 Série de transformations

Cette section présente les exemples de transformation que nous avons choisi d'analyser.

`T1` - Renommer la table `CLIENT` en `CUSTOMER`.
`T2` - Renommer la colonne `Adresse` de `CLIENT` en `NumRue`.
`T3` - Scinder la colonne `Adresse` de `CLIENT` en deux colonnes `Num` et `Rue`.
`T4` - Scinder la table `CLIENT` en créant une nouvelle table `ADRESSE` contenant la colonne `Adresse` de `CLIENT`.
`T5` - Fusionner les tables `CLIENT` et `ADRESSE` en une seule table `CLIENT` contenant toutes les colonnes des deux tables.

7.5 Série de requêtes

Cette section présente les requêtes que nous allons soumettre à chacune des transformations.

`R1` - Afficher toutes les informations des clients.

```
SELECT SQL_NO_CACHE NCli, Nom, Adresse, Localite, Cat, Compte
FROM CLIENT
order by NCli
```

R2 - Ajouter un nouveau client nommé "Mouse".

Pour faciliter l'automatisation des tests de performance, contrairement à ce que suggère la Figure 7.1, nous avons choisi de définir le type de la colonne identifiante `NCli` de la table `CLIENT` comme un nombre *auto-incremental*. Cela signifie qu'à chaque insertion dans cette table, MySQL génère un nombre unique en incrémentant la valeur du dernier identifiant, nombre qu'il va automatiquement attribuer au champ identifiant du nouveau client. Il est donc inutile de préciser la valeur de l'identifiant lors de l'insertion.

```
INSERT INTO CLIENT(Nom,Adresse,Localite,Cat,Compte)
VALUES  ('Mouse','1, avenue du parc','Chessy' ,'C1',1000.00)
```

R3 - Mettre à jour l'adresse du/des client(s) nommé(s) "Mouse".

```
UPDATE CLIENT
SET Adresse='EuroDisney',
    Localite='Paris'
WHERE Nom='Mouse''
```

R4 - Supprimer le/les client(s) nommé(s) "Mouse".

```
DELETE FROM CLIENT WHERE Nom = 'Mouse'
```

R5 - Agrandir la taille de la colonne `Nom` de la table `CLIENT` à 25 caractères.

```
ALTER TABLE CLIENT CHANGE Nom Nom char(25)
```

R6 - Ajouter un index sur l'adresse des clients.

```
CREATE INDEX KEYCLIADR ON CLIENT (Adresse)
```

7.6 Le traitement du couple (T1,R1) décrit étape par étape

Les résultats obtenus de cette étude sont issus d'un processus automatisé qui crée les tables, soumet les requêtes, et récolte les résultats. Cependant, par soucis de compréhension du lecteur, pour la première transformation `T1`, nous allons détailler le processus lié à la première requête `R1` sur une table contenant 10 lignes.

Avant transformation, le résultat de la requête R1 donnait le résultat suivant :

```
mysql> SELECT SQL_NO_CACHE NCli, Nom, Adresse, Localite, Cat, Compte
    -> FROM CLIENT
    -> order by NCli;
```

```
+------+-----------+-------------------+----------+------+----------+
| NCli | Nom       | Adresse           | Localite | Cat  | Compte   |
+------+-----------+-------------------+----------+------+----------+
|    1 | HANSENNE  | 23, a. Dumont     | Poitiers | C1   |  1250.00 |
|    2 | MERCIER   | 25, r. Lemaitre   | Namur    | C1   | -2300.00 |
|    3 | MONTI     | 112, r. Neuve     | Geneve   | B2   |     0.00 |
|    4 | TOUSSAINT | 5, r. Godefroid   | Poitiers | C1   |     0.00 |
|    5 | VANBIST   | 180, r. Florimont | Lille    | B1   |   720.00 |
|    6 | VANDERKA  | 3, av. des Roses  | Namur    | C1   | -4580.00 |
|    7 | GILLET    | 14, r. de l'Ete   | Toulouse | B1   | -8700.00 |
|    8 | GOFFIN    | 72, r. de la Gare | Namur    | B2   | -3200.00 |
|    9 | FERARD    | 65, r. du Tertre  | Poitiers | B2   |   350.00 |
|   10 | AVRON     | 8, ch. de la Cure | Toulouse | B1   | -1700.00 |
+------+-----------+-------------------+----------+------+----------+
10 rows in set (0.03 sec)
```

Suite à la transformation T1 et à l'utilisation de la vue suivante

```
CREATE VIEW CLIENT (NCli, Nom, Adresse, Localite, Cat, Compte) AS
  SELECT Ncli, Nom, Adresse, Localite, Cat, Compte
  FROM CUSTOMER
```

sans modification de la requête R1, le résultat de la requête est le suivant :

```
mysql> SELECT SQL_NO_CACHE NCli, Nom, Adresse, Localite, Cat, Compte
    -> FROM CLIENT
    -> order by NCli;
+------+-----------+-------------------+----------+------+----------+
| NCli | Nom       | Adresse           | Localite | Cat  | Compte   |
+------+-----------+-------------------+----------+------+----------+
|    1 | HANSENNE  | 23, a. Dumont     | Poitiers | C1   |  1250.00 |
|    2 | MERCIER   | 25, r. Lemaitre   | Namur    | C1   | -2300.00 |
|    3 | MONTI     | 112, r. Neuve     | Geneve   | B2   |     0.00 |
|    4 | TOUSSAINT | 5, r. Godefroid   | Poitiers | C1   |     0.00 |
|    5 | VANBIST   | 180, r. Florimont | Lille    | B1   |   720.00 |
|    6 | VANDERKA  | 3, av. des Roses  | Namur    | C1   | -4580.00 |
|    7 | GILLET    | 14, r. de l'Ete   | Toulouse | B1   | -8700.00 |
|    8 | GOFFIN    | 72, r. de la Gare | Namur    | B2   | -3200.00 |
|    9 | FERARD    | 65, r. du Tertre  | Poitiers | B2   |   350.00 |
|   10 | AVRON     | 8, ch. de la Cure | Toulouse | B1   | -1700.00 |
+------+-----------+-------------------+----------+------+----------+
10 rows in set (0.03 sec)
```

Suite à la transformation T1 et à la modification de R1 en

```
SELECT SQL_NO_CACHE NCli, Nom, Adresse, Localite, Cat, Compte
FROM CUSTOMER
order by NCli
```

sans utilisation d'une vue, le résultat de la requête sur la base de données après transformation est le suivant :

```
mysql> SELECT SQL_NO_CACHE NCli, Nom, Adresse, Localite, Cat, Compte
    -> FROM CUSTOMER
    -> order by NCli;


+------+-----------+-------------------+----------+------+----------+
| NCli | Nom       | Adresse           | Localite | Cat  | Compte   |
+------+-----------+-------------------+----------+------+----------+
|    1 | HANSENNE  | 23, a. Dumont     | Poitiers | C1   |  1250.00 |
|    2 | MERCIER   | 25, r. Lemaitre   | Namur    | C1   | -2300.00 |
|    3 | MONTI     | 112, r. Neuve     | Geneve   | B2   |     0.00 |
|    4 | TOUSSAINT | 5, r. Godefroid   | Poitiers | C1   |     0.00 |
|    5 | VANBIST   | 180, r. Florimont | Lille    | B1   |   720.00 |
|    6 | VANDERKA  | 3, av. des Roses  | Namur    | C1   | -4580.00 |
|    7 | GILLET    | 14, r. de l'Ete   | Toulouse | B1   | -8700.00 |
|    8 | GOFFIN    | 72, r. de la Gare | Namur    | B2   | -3200.00 |
|    9 | FERARD    | 65, r. du Tertre  | Poitiers | B2   |   350.00 |
|   10 | AVRON     | 8, ch. de la Cure | Toulouse | B1   | -1700.00 |
+------+-----------+-------------------+----------+------+----------+
10 rows in set (0.03 sec)
```

Nous appliquons la même démarche sur des tables CLIENT garnies de 100, 1000, 10000, et 100000 lignes ce qui nous permet de remplir le tableau suivant :

T1(Renommmer table)/R1(SELECT)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	31 ms	31 ms	31 ms
100	31 ms	30 ms	31 ms
1000	47 ms	48 ms	47 ms
10 000	141 ms	138 ms	146 ms
10 0000	813 ms	809 ms	828 ms

7.7 Résultats

Dans cette section, nous allons présenter les résultats obtenus durant cette étude de cas. Pour ce faire, à chacune des transformations T_X décrites à la Section 7.4, nous allons soumettre chacune des requêtes R_Y définies à la Section 7.5. Pour éviter de charger ce chapitre avec des tableaux de résultats, nous nous limiterons, pour chaque transformation, à :

1. rappeller sa définition
2. montrer la vue générée
3. réécrire les requêtes
4. illustrer une partie des résultats

Les temps d'exécution détaillés sont disponibles en annexe (Annexe A).

7.7.1 Transformation T1

7.7.1.1 Définition

T1 ≡ Renommer la table CLIENT en CUSTOMER

7.7.1.2 Vue générée

La vue générée pour s'adapter à la transformation T1 peut être créée comme suit :

```
CREATE VIEW CLIENT (NCli, Nom, Adresse, Localite, Cat, Compte) AS
  SELECT Ncli, Nom, Adresse, Localite, Cat, Compte
  FROM CUSTOMER
```

7.7.1.3 Réécriture des requêtes

R1 modifiée pour T1

```
SELECT SQL_NO_CACHE NCli, Nom, Adresse, Localite, Cat, Compte
FROM CUSTOMER
order by NCli
```

R2 modifiée pour T1

```
INSERT INTO CUSTOMER (Nom,Adresse,Localite,Cat,Compte)
VALUES ('Mouse','1, avenue du parc','Chessy' ,'C1',1000.00)
```

R3 modifiée pour T1

```
UPDATE CUSTOMER SET
  Adresse='EuroDisney',
  Localite='Paris'
WHERE Nom='Mouse'
```

R4 modifiée pour T1

```
DELETE FROM CUSTOMER WHERE Nom='Mouse'
```

R5 modifiée pour T1

```
ALTER TABLE CUSTOMER CHANGE Nom Nom char(25)
```

R6 modifiée pour T1

```
CREATE INDEX KEYCLIADR ON CUSTOMER(Localite)
```

7.7.1.4 Illustration d'une partie des résultats

Les Figures 7.2 et 7.3 illustrent sous forme graphique les résultats obtenus en appliquant la série de requêtes (Section 7.5) à une table garnie respectivement de 10 000 et de 100 000 lignes. Le détail des temps de réponse obtenus est disponible à l'Annexe A.

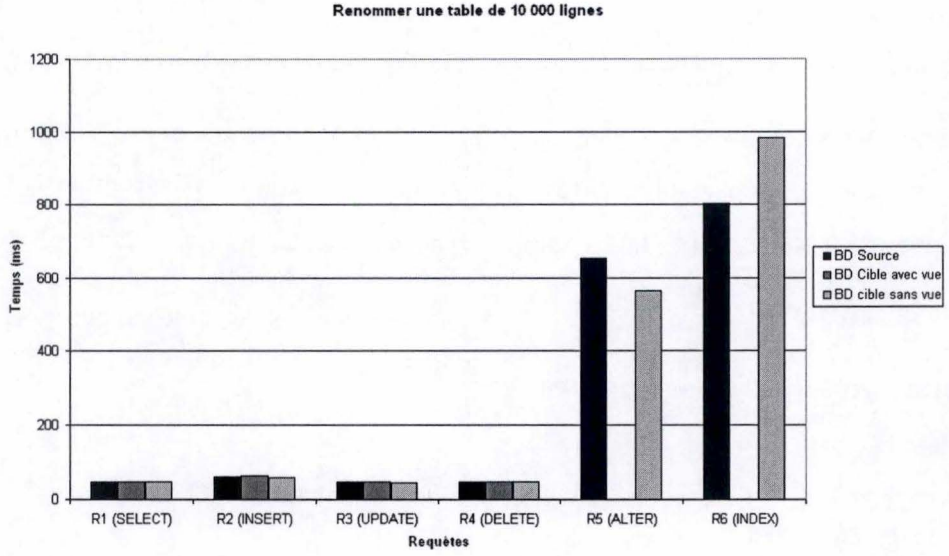


FIGURE 7.2 – *Illustration des résultats obtenus sur une table de 10 000 lignes*

Un temps d'exécution nul reflète l'occurrence d'une erreur lors de l'exécution de la requête. Voici les messages d'erreur envoyés par le script de soumission des requêtes :

- Lors de la modification de la taille de la colonne via la vue :
  `java.sql.SQLException : 'renameTableCbl10.client' is not BASE TABLE`

- Lors de l'ajout d'un index via la vue :
  `java.sql.SQLException : 'renameTableCbl10.client' is not BASE TABLE`

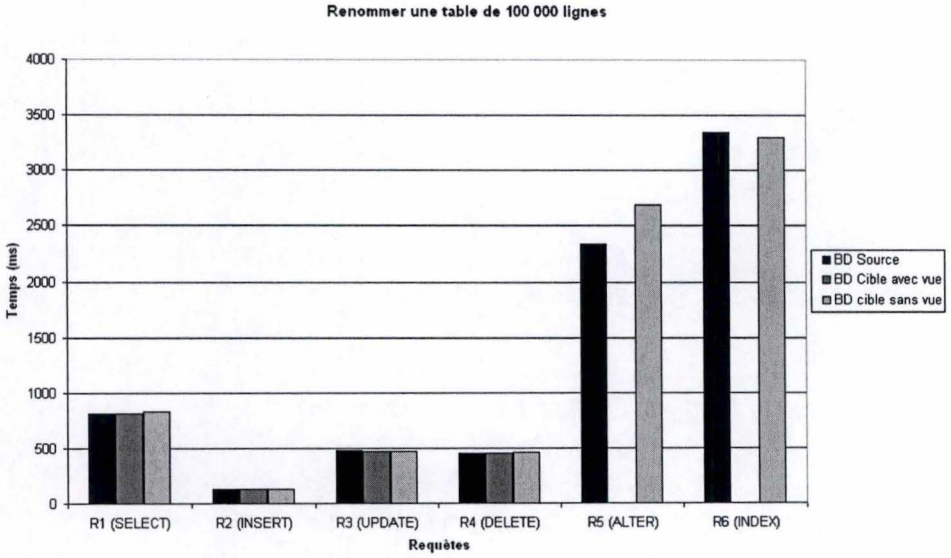


FIGURE 7.3 – *Illustration des résultats obtenus sur une table de 100 000 lignes*

7.7.2 Transformation T2

7.7.2.1 Définition

T2 ≡ Renommer la colonne `Adresse` de `CLIENT` en `NumRue`

7.7.2.2 Vue générée

La vue générée pour s'adapter à la transformation T2 peut être créée comme suit :

```
CREATE VIEW CLIENT (NCli, Nom, Adresse, Localite, Cat, Compte)
  AS SELECT NCli, Nom, NumRue, Localite, Cat, Compte
  FROM CLIENT_V1
```

7.7.2.3 Réécriture des requêtes

R1 modifiée pour T2

```
SELECT SQL_NO_CACHE NCli, Nom, NumRue, Localite, Cat, Compte
FROM CLIENT_V1
order by NCli";
```

R2 modifiée pour T2 :

```
INSERT INTO CLIENT_V1 (NOM,NUMRUE,LOCALITE,CAT,COMPTE)
VALUES ('Mouse','1, avenue du parc','Chessy','C1',1000.00)
```

R3 modifiée pour T2 :

```
UPDATE CLIENT_V1 SET
  NumRue='EuroDisney',
  Localite='Paris'
WHERE Nom='Mouse'
```

R4 modifiée pour T2 :

```
DELETE FROM CLIENT_V1 WHERE Nom='Mouse'
```

R5 modifiée pour T2 :

```
ALTER TABLE CLIENT_V1 CHANGE Nom Nom char(25)
```

R6 modifiée pour T2 :

```
CREATE INDEX KEYCLIADR ON CLIENT_V1(Localite)
```

7.7.2.4 Illustration d'une partie des résultats

Les Figures 7.4 et 7.5 illustrent sous forme graphique les résultats obtenus en appliquant la série de requêtes à une table garnie respectivement de 10 000 et de 100 000 lignes. Le détail des temps de réponse obtenus est disponible à l'Annexe A.

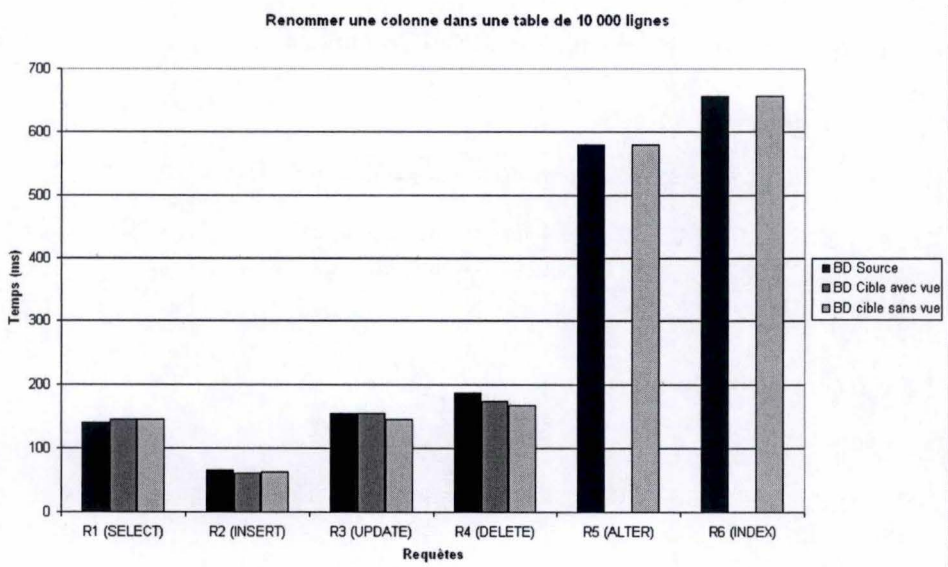


FIGURE 7.4 – *Illustration des résultats obtenus sur une table de 10 000 lignes*

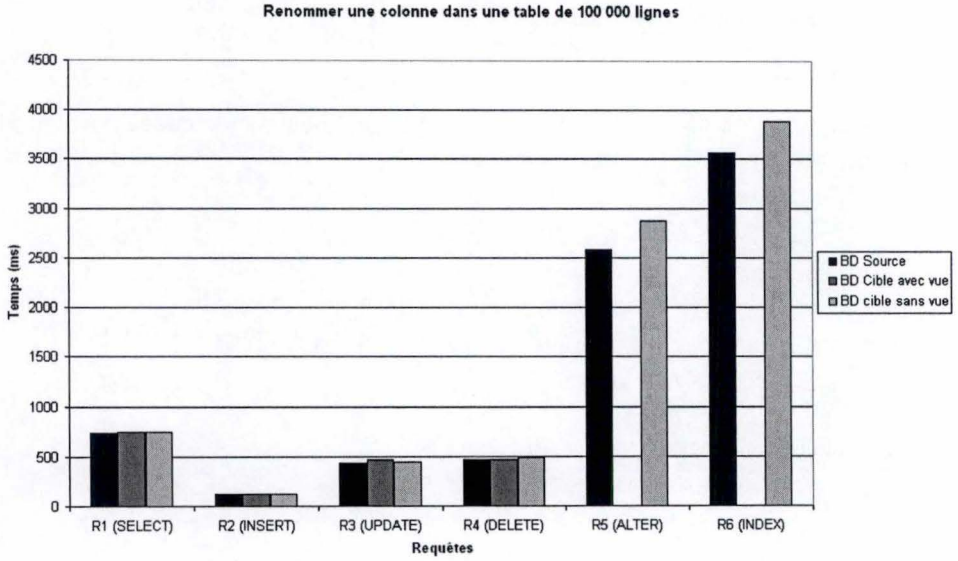


FIGURE 7.5 – *Illustration des résultats obtenus sur une table de 100 000 lignes*

Un temps d'exécution nul reflète l'occurrence d'une erreur lors de l'exécution de la requête. Voici les messages d'erreur envoyés par le script de soumission des requêtes :

- Lors de la modification de la taille de la colonne via la vue :
  `java.sql.SQLException : 'renameColumnCbl10.client' is not BASE TABLE`

- Lors de l'ajout d'un index via la vue :
  `java.sql.SQLException : 'renameColumnCbl10.client' is not BASE TABLE`

7.7.3 Transformation T3

7.7.3.1 Définition

T3 ≡ Scinder la colonne `Adresse` de `CLIENT` en deux colonnes `Num` et `Rue`

7.7.3.2 Vue générée

La vue générée pour s'adapter à la transformation T3 peut être créée comme suit :

```
CREATE VIEW CLIENT (NCli, Nom, Adresse, Localite, Cat, Compte) AS
  SELECT NCli, Nom, concat(Num,Rue), Localite, Cat, Compte
  FROM CLIENT_V1
```

7.7.3.3 Réécriture des requêtes

R1 modifiée pour T3 :

```
SELECT SQL_NO_CACHE NCli, Nom, concat(Num,Rue) AS Adresse, Localite, Cat, Compte
FROM CLIENT_V1
order by NCli
```

R2 modifiée pour T3 :

```
INSERT INTO CLIENT_V1 (NOM,NUM,RUE,LOCALITE,CAT,COMPTE)
VALUES ('Mouse','1',', avenue du parc','Chessy','C1',1000.00)
```

R3 modifiée pour T3 :

```
UPDATE CLIENT_V1 SET
  Num='',
  Rue='EuroDisney',
  Localite='Paris'
  WHERE Nom='Mouse'
```

R4 modifiée pour T3 :

```
DELETE FROM CLIENT_V1 WHERE Nom='Mouse'
```

R5 modifiée pour T3 :

```
ALTER TABLE CLIENT_V1 CHANGE Nom Nom char(25)
```

R6 modifiée pour T3 :

```
CREATE INDEX KEYCLIADR ON CLIENT_V1(Localite)
```

7.7.3.4 Illustration d'une partie des résultats

Les Figures 7.6 et 7.7 illustrent sous forme graphique les résultats obtenus en appliquant la série de requêtes (Section 7.5) à une table garnie respectivement de 10 000 et de 100 000 lignes. Le détail des temps de réponse obtenus est disponible à l'Annexe A.

Un temps d'exécution nul reflète l'occurrence d'une erreur lors de l'exécution de la requête. Voici les messages d'erreur envoyés par le script de soumission des requêtes :

- Lors de l'insertion du nouveau client via la vue :
  `java.sql.SQLException : The target table CLIENT of the INSERT is not insertable-into`

- Lors de la mise à jour du client via la vue, nous obtenons l'erreur suivante :
  `java.sql.SQLException : Column 'Adresse' is not updatable`

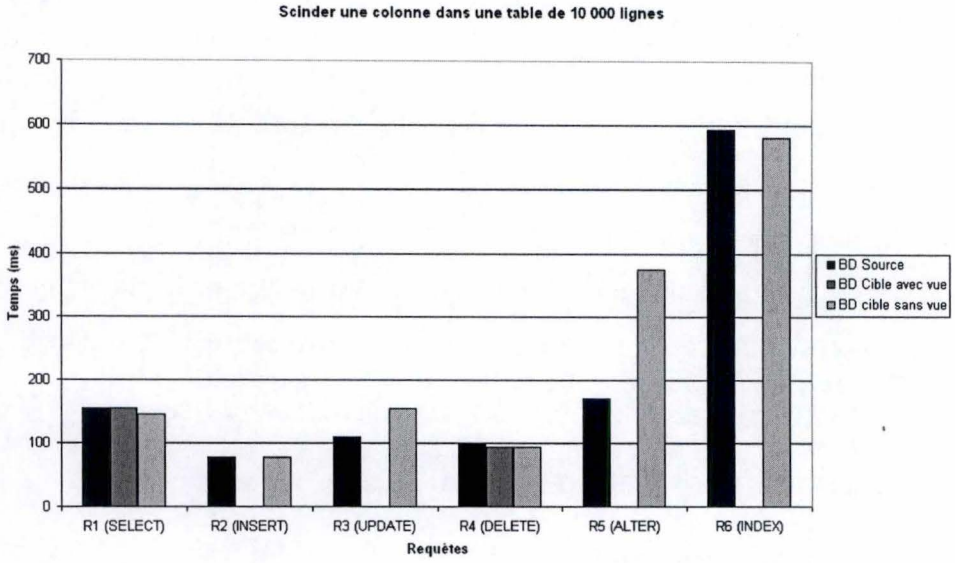


FIGURE 7.6 – *Illustration des résultats obtenus sur une tables de 10 000 lignes*

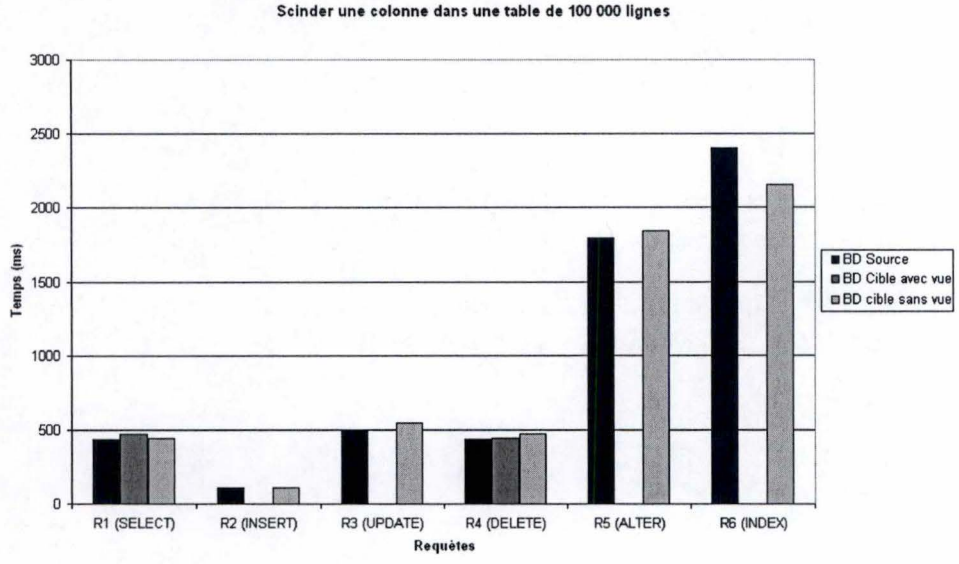


FIGURE 7.7 – *Illustration des résultats obtenus sur une table de 100 000 lignes*

- Lors de la modification de la taille de la colonne via la vue :
  `java.sql.SQLException : 'splitColumnCbl10.client' is not BASE TABLE`

- Lors de l'ajout d'un index via la vue :
  `java.sql.SQLException : 'splitColumnCbl10.client' is not BASE TABLE`

7.7.4 Transformation T4

7.7.4.1 Définition

T4 ≡ Scinder la table CLIENT en créant une nouvelle table ADRESSE contenant la colonne Adresse de CLIENT

7.7.4.2 Vue générée

La vue générée pour s'adapter à la transformation T4 peut être créée comme suit :

```
CREATE VIEW CLIENT (NCli, Nom, Adresse, Localite, Cat, Compte) AS
SELECT ADRESSE.NCli, CLIENT_V1.Nom, ADRESSE.Adresse,
       CLIENT_V1.Localite, CLIENT_V1.Cat, CLIENT_V1.Compte
FROM ADRESSE, CLIENT_V1
WHERE ADRESSE.NCli = CLIENT_V1.NCli
```

7.7.4.3 Réécriture des requêtes

R1 modifiée pour T4 :

```
SELECT SQL_NO_CACHE CLIENT_V1.NCli, Nom, Adresse, Localite, Cat, Compte
FROM CLIENT_V1, ADRESSE
WHERE CLIENT_V1.NCli=ADRESSE.NCli
order by CLIENT_V1.NCli
```

R2 modifiée pour T4 :

```
INSERT INTO CLIENT_V1 (Nom,Localite,Cat,Compte)
VALUES ('Mouse','Chessy','C1',1000.00)";

INSERT INTO ADRESSE (NCli, Adresse)
VALUES (LAST_INSERT_ID(),'1, avenue du parc')
```

R3 modifiée pour T4 :

```
UPDATE CLIENT_V1 SET
  Localite='Paris'
WHERE Nom='Mouse'

UPDATE ADRESSE SET
  Adresse='EuroDisney'
WHERE Ncli in(
  Select NCli
  FROM CLIENT_V1
  WHERE Nom='Mouse')
```

R4 modifiée pour T4 :

```
DELETE FROM ADRESSE WHERE NCli in (
  SELECT NCli
  FROM CLIENT
  WHERE Nom='Mouse')

DELETE FROM CLIENT_V1 WHERE Nom='Mouse'
```

R5 modifiée pour T4 :

```
ALTER TABLE CLIENT_V1 CHANGE Nom Nom char(25)
```

R6 modifiée pour T4 :

```
CREATE INDEX KEYCLIADR ON CLIENT_V1(Localite)
```

7.7.4.4 Illustration d'une partie des résultats

Les Figures 7.8 et 7.9 illustrent sous forme de graphique les résultats obtenus en appliquant la série de requêtes (Section 7.5) à une table garnie respectivement de 10 000 et de 100 000 lignes. Le détail des temps de réponse obtenus est disponible à l'Annexe A.

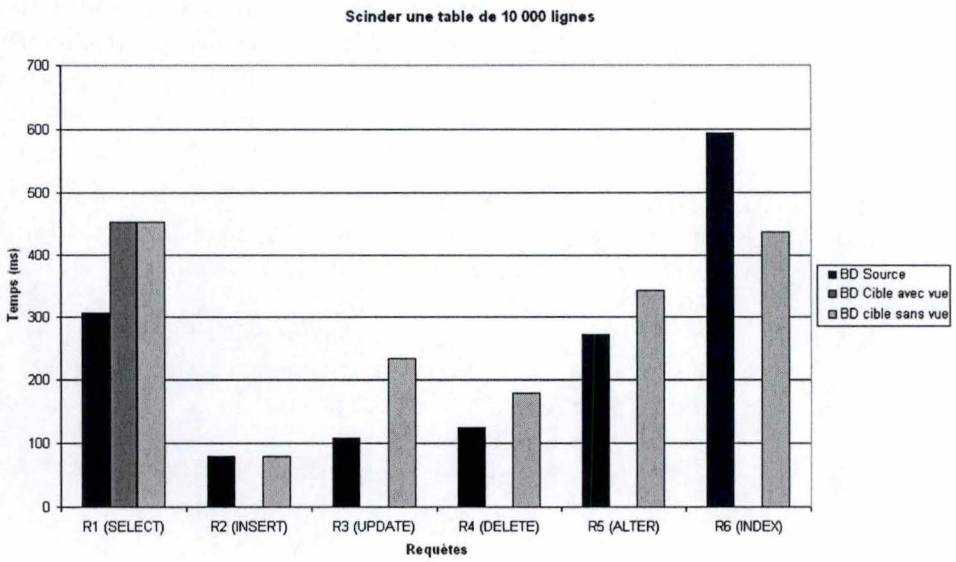


FIGURE 7.8 – *Illustration des résultats obtenus sur une tables de 10 000 lignes*

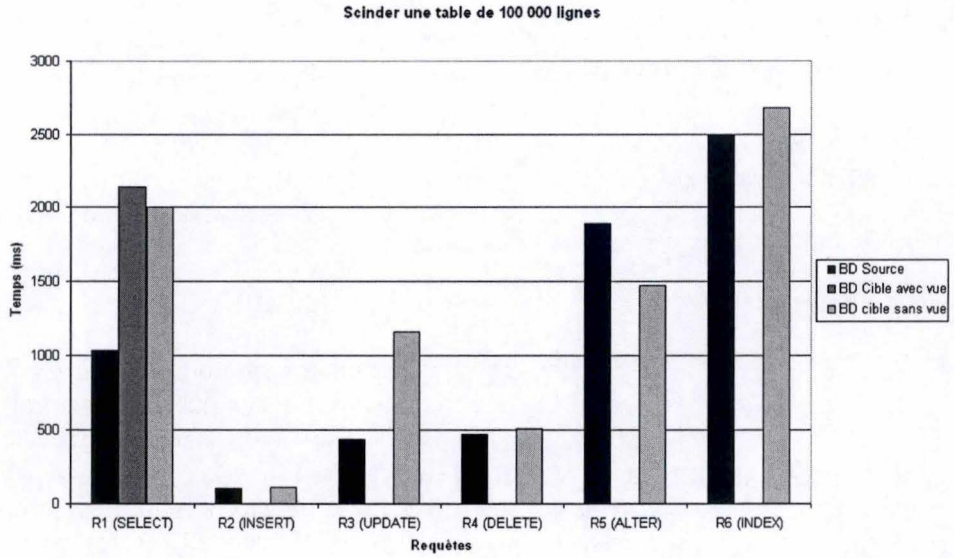


FIGURE 7.9 – *Illustration des résultats obtenus sur une table de 100 000 lignes*

Un temps d'exécution nul reflète l'occurrence d'une erreur lors de l'exécution de la requête. Voici les messages d'erreur envoyés par le script de soumission des requêtes :

- Lors de l'insertion du nouveau client via la vue :
  `java.sql.SQLException : Can not modify more than one base table through a join view 'splitTableCbl10.client'`

- Lors de la mise à jour du client via la vue, nous obtenons l'erreur suivante :
  `java.sql.SQLException : Can not modify more than one base table through a join view 'splitTableCbl10.client'`

- Lors de la suppression du/des client(s) via la vue :
  `java.sql.SQLException : Can not delete from join view 'splitTableCbl10.client'`

- Lors de la modification de la taille de la colonne via la vue :
  `java.sql.SQLException : 'splitTableCbl10.client' is not BASE TABLE`

- Lors de l'ajout d'un index via la vue :
  `java.sql.SQLException : 'splitTableCbl10.client' is not BASE TABLE`

7.7.5 Transformation T5

7.7.5.1 Définition

T5 ≡ Fusionner les tables CLIENT et ADRESSE en une seule table CLIENT contenant toutes les colonnes des deux tables

7.7.5.2 Vue générée

Les vues générées pour s'adapter à la transformation T5 peuvent être créées comme suit :

```
CREATE VIEW CLIENT (NCli, Nom, Localite, Cat, Compte)
  AS SELECT NCli, Nom, Localite, Cat, Compte
  FROM CLIENT_V1

CREATE VIEW ADRESSE (NCli, Adresse)
  AS SELECT NCli, Adresse
  FROM CLIENT_V1
```

7.7.5.3 Réécriture des requêtes

7.7.5.4 Requête R1

Dans le cas de la transformation T5, la requête R1 de base est :

```
SELECT ADRESSE.NCli, CLIENT.Nom, ADRESSE.Adresse,
       CLIENT.Localite, CLIENT.Cat, CLIENT1.Compte
FROM ADRESSE, CLIENT
WHERE ADRESSE.NCli = CLIENT.NCli
```

et la requête R1 modifiée devient :

```
SELECT SQL_NO_CACHE NCli, Nom, Adresse, Localite, Cat, Compte
FROM CLIENT_V1
order by NCli
```

La requête R2 de base est :

```
INSERT INTO CLIENT (Nom,Localite,Cat,Compte)
VALUES ('Mouse','Chessy','C1',1000.00)

INSERT INTO ADRESSE (NCli, Adresse)
VALUES (LAST_INSERT_ID(),'1, avenue du parc')
```

où LAST_INSERT_ID() est le dernier identifiant auto-incrémental généré par le dernier INSERT et la requête R2 modifiée devient :

```
INSERT INTO CLIENT_V1 (Nom,Adresse,Localite,Cat,Compte)
VALUES  ('MOUSE','1, avenue du parc','Chessy' ,'C1',1000.00)
```

La requête R3 de base est :

```
UPDATE CLIENT SET
  Localite='Paris'
WHERE Nom='Mouse'

UPDATE ADRESSE SET
  Adresse='EuroDisney'
WHERE Ncli in(
  Select NCli
  FROM CLIENT
  WHERE Nom='Mouse')
```

et la requête R3 modifiée devient :

```
UPDATE CLIENT_V1
SET Adresse='EuroDisney',
    Localite='Paris'
WHERE Nom='Mouse'
```

La requête R4 de base est :

```
DELETE FROM CLIENT WHERE Nom='Mouse'
```

et la requête R4 modifiée devient :

```
DELETE FROM CLIENT_V1 WHERE Nom='Mouse'
```

R5 modifiée pour T5 :

```
ALTER TABLE CLIENT_V1 CHANGE Nom Nom char(25)
```

R6 modifiée pour T5 :

```
CREATE INDEX KEYCLIADR ON CLIENT_V1(Localite)
```

7.7.5.5 Illustration d'une partie des résultats

Les Figures 7.10 et 7.11 illustrent sous forme graphique les résultats obtenus en appliquant la série de requêtes (Section 7.5) à une table garnie respectivement de 10 000 et de 100 000 lignes. Le détail des temps de réponse obtenus est disponible à l'Annexe A.

Un temps d'exécution nul reflète l'occurrence d'une erreur lors de l'exécution de la requête. Voici les messages d'erreur envoyés par le script de soumission des requêtes :

- Lors de l'insertion du nouveau client via la vue :
  `java.sql.SQLException : Field of view 'mergeTableCbl10.client' underlying table doesn't have a default value`

- Lors de la mise à jour du client via la vue, nous obtenons l'erreur suivante :
  `java.sql.SQLException : The definition of table 'CLIENT' prevents operation UPDATE on table 'ADRESSE'`

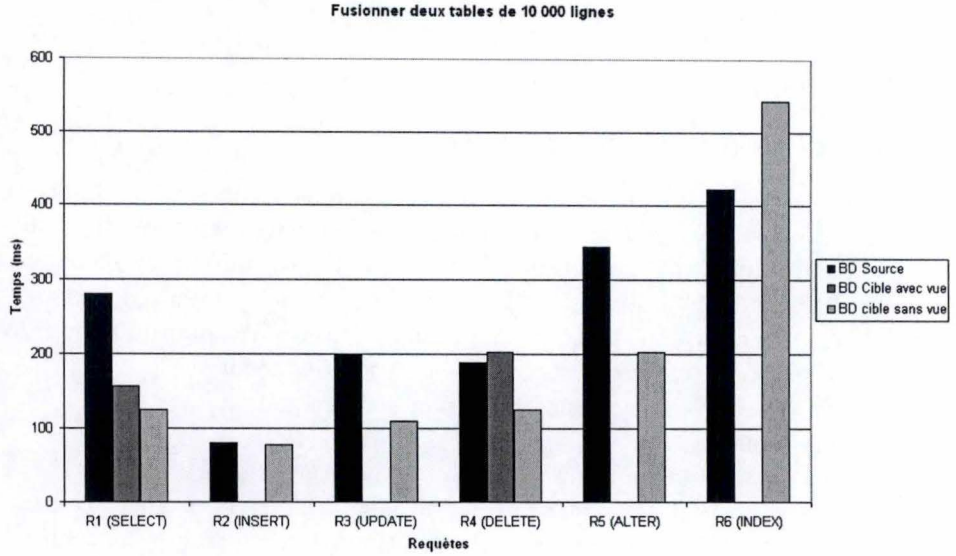


FIGURE 7.10 – *Illustration des résultats obtenus sur une tables de 10 000 lignes*

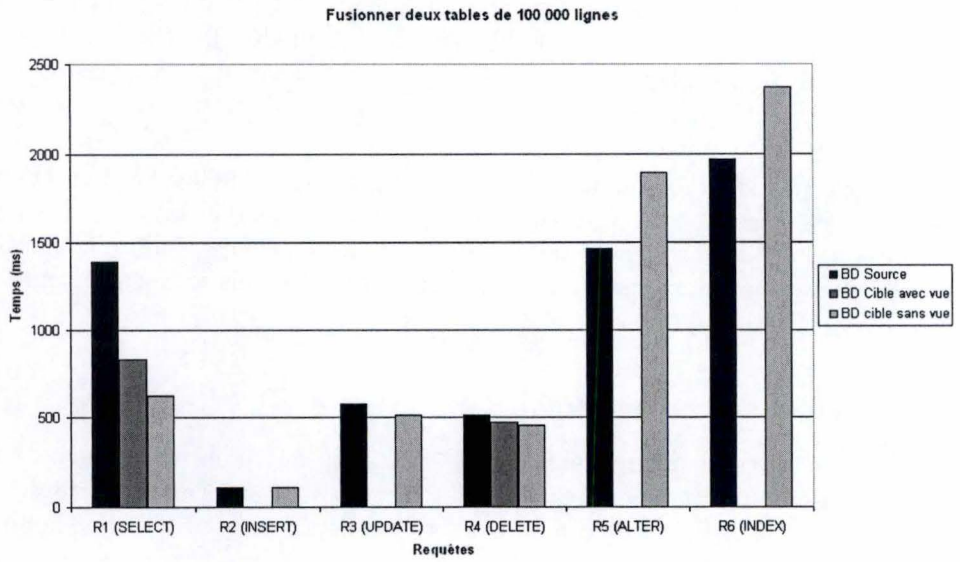


FIGURE 7.11 – *Illustration des résultats obtenus sur une table de 100 000 lignes*

- Lors de la modification de la taille de la colonne via la vue :
  `java.sql.SQLException : 'mergeTableCbl10.client' is not BASE TABLE`

- Lors de l'ajout d'un index via la vue :
  `java.sql.SQLException : 'mergeTableCbl10.client' is not BASE TABLE`

7.8 Discussion

7.8.1 Critique de l'environnement de test

7.8.1.1 Temps d'exécution des requêtes

Tout d'abord nous tenons à signaler que les temps obtenus ne reflètent pas exactement les temps d'exécution des requêtes. Le client MySQL possède une fonction lui permettant de calculer le temps d'exécution d'une requête au centième de seconde près. Malheureusement, cette fonction est liée au client et pas au serveur, il nous semble donc difficile de l'exploiter dans un script d'automatisation. C'est pourquoi, nous avons procédé comme suit pour calculer le temps d'exécution des requêtes :
- Enregistrement du timestamp [1] avant exécution de la requête
- Exécution de la requête
- Enregistrement du timestamp après exécution de la requête
- Le temps d'éxecution = timestamp après exécution - timestamp avant exécution

Il y a donc un temps additionnel au temps d'exécution de la requête dû au temps CPU de deux appels à la fonction retournant le timestamp et aussi au temps de l'opération de soustraction.

De plus, malgré le fait que nous utilisons l'option `SQL_NO_CACHE` [2] dans nos requêtes, les exécutions successives d'une requête sont de plus en plus rapides. Les conseils de gestion de la cache de MySQL expliqué dans [Sch05] n'y ont rien changé... Pour contourner ce problème, nous avons pris soin de redémarrer le serveur MySQL après chaque requête sur une même base de données.
Ce manque de précision du temps d'exécution des requêtes ne semble pas poser de problème dans la mesure où le but de cette étude est de détecter d'éventuels gains/pertes de performance, et pas de prendre des mesures exactes de temps. La seule contrainte que nous devons nous fixer étant d'exécuter tous nos tests dans les mêmes conditions d'utilisation du CPU et du serveur MySQL.

7.8.1.2 Résultats des requêtes

Lorsque nous soumettons une requête de type `SELECT` sur une table ne contenant que 10 lignes, il est aisé de comparer les résultats obtenus avant transformation, après transformation avec/sans vue(s) et de constater s'ils sont identiques. Une fois que l'on passe à 100, 1000, 10000, et 100000 lignes, la comparaison est moins aisée... Rien n'assure que les résultats obtenus sont bien ceux attendus. Le nombre de lignes de résultat retourné par la requête peut déjà constituer un premier indicateur. Si ce nombre était celui attendu, nous avons considéré que le résultat obtenu était correct. Idéalement, il aurait fallu stocker temporairement le résultat de chacune des requêtes dans une table, et ensuite comparer le contenu de ces deux tables.

1. Le nombre de secondes écoulées depuis le 01/01/1970 à 00h00
2. Option MySQL étant censée éviter la mise en cache de la requête

7.8.1.3 Limitations

Cette étude de cas a été réalisée à l'aide du SGBD MySQL. Tous les SGBD n'implémentant pas les vues de la même manière, nous ne prétendons pas tirer de conclusions générales applicables à tous les SGBD. Mais, certains résultats obtenus interpellent et amènent à une discussion sur l'utilisation des vues pour l'évolution des bases de données.

7.8.2 Discussion des résultats

Les résultats obtenus ci-dessus confirment ce qui avait été dit au Chapitre 1, à savoir que la création d'index sur une vue n'est pas autorisée. De plus, on constate qu'il est généralement impossible de modifier la structure d'une table de base à travers une vue.

Commençons cette discussion par un tableau récapitulatif présentant le support ou non d'un type de requête par une nature de transformation.

	SELECT	INSERT	UPDATE	DELETE
Renommer une table	✓	✓	✓	✓
Renommer une colonne	✓	✓	✓	✓
Scinder une colonne	✓	X	X	✓
Scinder une table	✓	X	X	X
Fusionner deux tables	✓	X	X	✓

L'utilisation d'une vue pour les transformations de ***renommage*** supporte les quatre types de requêtes `SELECT-INSERT-UPDATE-DELETE`. Les temps de réponse des requêtes suite à de telles transformations sont similaires que l'on utilise une vue ou que l'on modifie la requête. Cependant, suivant la nature du code applicatif, il nous semble plus judicieux de modifier les requêtes. En effet, cette modification nécessite juste de remplacer l'ancien de nom de l'objet par le nouveau nom. La plupart des éditeurs de texte proposent une fonction "Chercher-Remplacer" qui permet à l'utilisateur d'indiquer une chaîne de caractères à remplacer par une autre, le reste se faisant de manière automatique. Ceci évitera de devoir renommer la table faisant l'objet de la transformation, d'ajouter une vue et de risquer des erreurs en cas de création d'index.

Lorsque la transformation consiste à ***scinder une colonne***, il est alors impossible d'insérer une nouvelle ligne d'une table via la vue. En effet, les anciennes requêtes d'insertion souhaitent ajouter une valeur qu'il faudrait maintenant répartir dans deux colonnes. Le SGBD ne sachant pas où scinder l'information pour l'insérer correctement dans chacune des deux colonnes, envoie une erreur. Pour le cas de la mise à jour d'une ligne, c'est moins univoque. Si les colonnes à mettre à jour ne comprennent pas celle qui a été précédement scindée, la mise à jour ne pose pas de problème. Comme c'était déjà le cas lors des transformations de renommage, quand la vue le permet, les temps d'exécution des requêtes sont du même ordre que ceux de la solution qui consiste à ré-écrire la requête. Nous recommandons donc d'utiliser la vue quand la table ayant

fait l'objet de la transformation n'est pas soumise à des ajouts de lignes et/ou quand la valeur de la colonne scindée n'est plus modifiée une fois insérée.

Comme nous pouvions nous y attendre, lorsque ***nous scindons la table en deux tables***, seule l'utilisation des requêtes de type `SELECT` est permise via la vue. Nous avons présenté au Chapitre 1 Section 1.5 les conditions nécessaires pour avoir une vue actualisable, et dans ce cas-ci, la vue étant construite sur une jointure, ces conditions ne sont pas remplies. Il est donc impossible de lui soumettre des requêtes de type `INSERT`, `UPDATE` et `DELETE`. Il nous reste la question des requêtes de type `SELECT`. On voit une forte augmentation du temps de réponse entre la situation avant transformation et la situation après transformation. Pour illustrer cette différence de temps d'exécution, nous allons appliquer la transformation à une table contenant de 10000, 20000, ..., 100000 lignes et reporter les résultats sur le graphique présenté à la Figure 7.12.

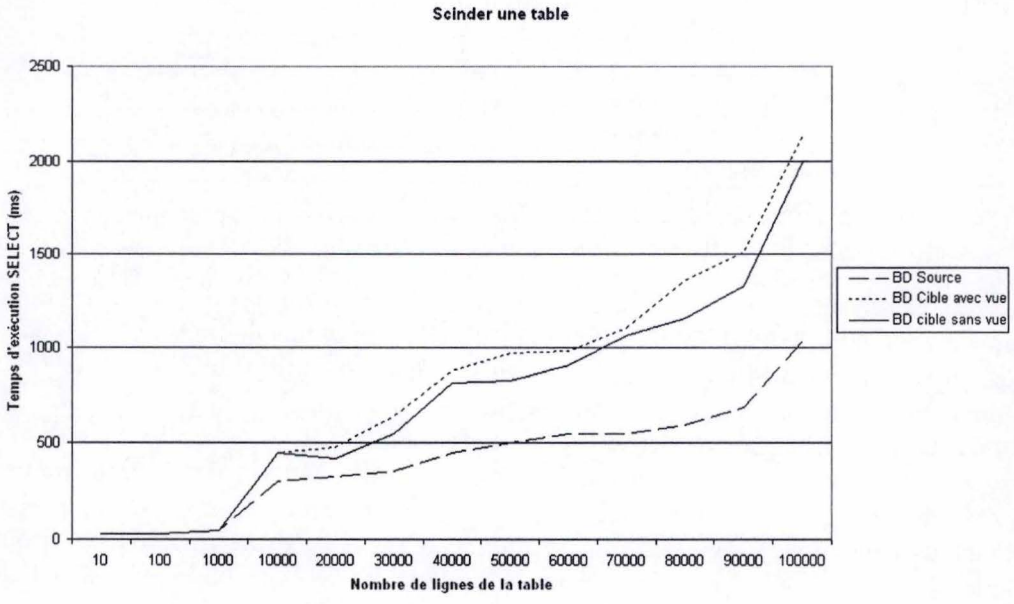


FIGURE 7.12 – *Illustration du temps d'éxecution d'un SELECT avant et après avoir scindé une table en fonction du volume de données*

Nous remarquons qu'à partir de 1000 lignes la requête `SELECT` met plus de temps à s'exécuter qu'avant la transformation, cette augmentation de temps s'accentue avec le nombre de lignes à traiter. Ceci s'explique simplement par le fait qu'avant la transformation, une seule table était interrogée, alors que maintenant, suite à la transformation, l'équivalent de la requête contient une jointure.
Si l'on compare les temps d'exécution après transformation avec ou sans l'utilisation d'une vue, on remarque qu'ils sont du même ordre de grandeur. Il est dès lors opportun

de se poser la question de savoir quelle solution choisir... Etant donné la complexité de ré-écriture des requêtes, la vue peut permettre un gain de temps non négligeable, mais uniquement si le nombre de requêtes de type `SELECT` comparé au nombre total de requêtes exploitant cette table est significatif. En effet, il sera toujours nécessaire de modifier les requêtes de type `INSERT`, `DELETE`, `UPDATE`, et la création d'index sera impossible.
Il est important de signaler que sans la clause `order by`, l'ordre des lignes du résultat après transformation est différent de celui obtenu avant transformation.

Lors ***la fusion de deux tables***, seules les opérations de type `SELECT` et `DELETE` sont permises via la/les vue(s). L'impossibilité d'ajout (`INSERT`) ou de mise à jour (`UPDATE`) d'une ligne via une des deux vues s'explique dans notre étude de cas par le fait que les colonnes de la table de base non contenues dans la vue faisant l'objet de la requête n'avaient pas de valeur par défaut. En ce qui concerne les requêtes de type `SELECT`, on observe une forte diminution du temps de réponse entre la situation avant transformation et la situation après transformation. Pour illustrer cette différence de temps d'éxécution, nous allons appliquer la transformation à une table contenant de 10000, 20000, ..., 100000 lignes et reporter les résultats sur le graphique présenté à la Figure 7.13.

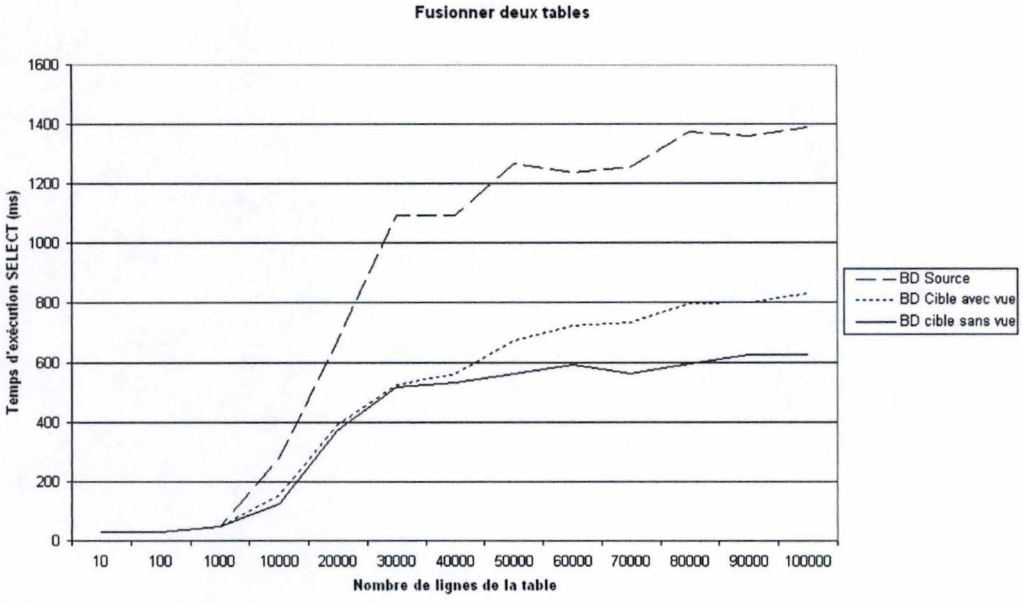


FIGURE 7.13 – *Illustration du temps d'éxecution d'un SELECT avant et après avoir fusionnéé deux tables en fonction du volume de données*

Nous remarquons qu'à partir de 1000 lignes la requête `SELECT` met moins de temps

à s'exécuter qu'avant la transformation, cette diminution de temps s'accentue avec le nombre de lignes à traiter. Ceci s'explique simplement par le fait qu'avant la transformation, deux tables étaient intérrogées, et qu'une jointure était donc nécessaire. Alors que suite à la transformation, une seule table est interrogée, supprimant le besoin d'une jointure.
Si l'on compare les temps d'exécution après transformation avec ou sans l'utilisation de vues, on remarque qu'à partir de 30000 lignes la solution basée sur les vues est plus lente, et que cet écart grandi avec le nombre de lignes à traiter. Ceci s'explique par le fait que les vues simulent deux tables qui sont interrogées via une jointure. Il est dès lors opportun de se poser la question de savoir quelle solution choisir... Etant donné la complexité de ré-écriture des requêtes, les vues peuvent permettre un gain de temps non négligeable, mais uniquement si le nombre de lignes de la table n'est pas trop important, et si le nombre de requêtes de type `SELECT` comparé au nombre total de requêtes exploitant cette table est significatif. En effet, si les vues sont définies sur des colonnes n'ayant pas de valeurs par défaut, il sera toujours nécessaire de modifier les requêtes de type `INSERT`, et `UPDATE`. De plus, la création d'index sera impossible.

Quatrième partie

Conclusion

Conclusion

Problématique et question générale de recherche

Au cours de sa vie, une base est amenée à être modifiée pour répondre à l'évolution des besoins des utilisateurs. Cette évolution se traduit par la modification de la structure de la base de données, des données, et des programmes exploitant ces données. Notre intérêt se situe au niveau du maintien de la compatibilité du code applicatif.

Une fois la structure de données modifiée, et les données migrées, il reste à l'analyste à adapter les programmes. Malheureusement, malgré de nombreux travaux traitant de cette problématique, peu d'entre eux proposent un outil d'aide à la réalisation de cette tâche. Pour pallier à ce manque, nous avons proposé une méthodologie de génération automatique de vues dans le but de définir l'ancienne structure de la base de données comme une vue de la nouvelle structure (*backward views*), permettant ainsi une meilleure compatibilité des anciennes requêtes. Nous avons ensuite mis en pratique notre méthodologie en implémentant un plug-in pour DB-Main en ayant pris soin, au préalable, de définir formellement les règles de génération.

Pertinence théorique et pratique du travail réalisé

L'utilisation des vues pour définir l'ancienne structure de la base de données comme une vue de la nouvelle structure permet une meilleure compatibilité des anciennes requêtes. De plus, cette approche axée sur la base de données isole l'applicatif et évite ainsi les difficultés liées à la modification du programme, souvent source d'erreurs .

Pour mettre en oeuvre cette approche, nous avons développé une méthodologie qui exploite une série de règles formelles de génération de vues. La généralisation de ces règles ne suffisant pas à aider l'analyste dans sa tâche d'adaptation des programmes, nous les avons implémentées au sein d'un plug-in DB-Main. Il en résulte un générateur automatique de vues dépendantes de la nature de la transformation.

A l'aide d'une étude de cas, nous avons tenté de démontrer que cette méthode est opérationnelle et efficace.

Nous avons conclu qu'elle a ses limites qui sont en grande partie dûes aux conditions nécessaires pour avoir une vue actualisable (Chapitre 1 Section 1.5). Plus particulièrement celle qui impose qu'il exite une correspondance ligne à ligne entre les colonnes de la vue et celles de la table de base, et celle qui impose que la vue ne référence pas plus d'une table dans sa clause `FROM`.
Hormis pour les transformations qui consistent au renommage d'un objet, ces limites ont pour conséquence de réduire l'interêt de l'utilisation de la vue à l'amélioration de la compatibilité des opérations de consultation (`SELECT`) et parfois de suppression (`DELETE`) des données. Les autres types d'opérations (`INSERT`,`UPDATE`) aboutissent à une erreur. Cette solution n'est donc qu'une solution partielle au problème de propagation de la modification de la base de données aux programmes.
Les temps d'exécution des requêtes sont quant à eux équivalents que l'on opte pour l'utilisation d'une vue *backward* ou pour la ré-écriture des requêtes, ceci quelque soit la nature de la transformation. Cependant, sans l'avoir testé, on peut se douter qu'au cours du temps, la formation d'une trop longue chaîne de vues pourrait avoir des répercussions négatives sur les performances.

Il est donc naïf de penser que cette approche est la solution optimale à l'évolution des programmes. Cependant, cela nous semble adapté à des périodes de transition durant lesquelles le programmeur réécrirait les requêtes. Cela permettrait de minimiser le temps d'indisponibilité de l'application.

Nouvelles pistes de recherche

Cette approche pourrait être encore approfondie, elle mène aussi à de nouvelles réflexions qui pourraient être sujettes à études...

Nous pourrions continuer sur cette voie et enrichir le catalogue des transformations prises en charge par notre méthodologie, cela en formalisant de nouvelles règles de génération de vues.

Les limitations liées aux performances et à la création d'index peuvent être supprimées par l'utilisation de vues matérialisées. Alors que les limitations dûes aux vues non actualisables peuvent être levées grâce à l'utilisation de "trigger `INSTEAD OF`". Les triggers sont des mécanismes constitués de code éxécutés sous certaines conditions. Les triggers `INSTEAD OF` sont des triggers qui décrivent la manière d'exécuter une requête de type [`INSERT|UPADTE|DELETE`] sur une vue qui n'est normalement pas actualisable. Ils semblent combler une grosse lacune de la solution proposée dans ce mémoire, il serait donc intéressant d'analyser leur fonctionnement et peut-être, si cela est possible, de les générer automatiquement.

Pour être complet concernant de l'utilisation des vues pour l'évolution des bases de données, il serait intéressant d'analyser la solution des {***forward views***}, qui consiste à conserver la structure du schéma d'origine et à utiliser une ou plusieurs vues de manière à simuler les transformations de la base de données.

Si l'on opte pour l'utilisation des vues comme solution temporaire pour minimiser le temps d'indisponibilité des programmes, il serait intéressant de voir dans quelle mesure on pourrait guider le programmeur dans la réécriture de ses requêtes. Les règles de génération de vues formalisées dans le cadre de ce mémoire pourraient servir à cette approche. On pourrait utiliser la clause `SELECT` de la vue et l'adapter légèrement de manière à remplacer le contenu de la clause `FROM` de la requête à modifier.
Supposons que l'on ait renommé la colonne `Adresse` de la table `CLIENT` en `NumRue` :
On pourrait, par exemple, réécrire la requête
- `SELECT Nom, Adresse, Localite`
- `FROM CLIENT`

comme suit
- `SELECT Nom, Adresse, Localite`
- `FROM (`***`SELECT Nom, NumRue`*** `as Adresse,` ***`Localite FROM CLIENT`***`)`

où ce qui figure en ***gras-italique*** dans la clause `FROM` est la clause `SELECT` de la vue générée pour minimiser l'impact de cette transformation.

Bibliographie

[AB08] MySQL AB. *MySQL 5.1 Reference Manual*, 2008.

[AS06] Scott W. Ambler and Pramodkumar J. Sadalage. *Refactoring Databases : Evolutionary Database Design.* Addison-Wesley, 2006.

[Cel95] Joe Celko. *Joe Celko's SQL for Smarties : Advanced SQL Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995.

[CH06] Anthony Cleve and Jean-Luc Hainaut. Co-transformations in database applications evolution. In *Generative and Transformational Techniques in Software Engineering*, pages 409–421, 2006.

[CMTZ08] Carlo Curino, Hyun J. Moon, Letizia Tanca, and Carlo Zaniolo. Schema evolution in wikipedia - toward a web information system benchmark. In José Cordeiro and Joaquim Filipe, editors, *International Conference on Enterprise Information Systems*, pages 323–332, 2008.

[CMZ08] Carlo A. Curino, Hyun J. Moon, and Carlo Zaniolo. Graceful database schema evolution : the prism workbench. *Proc. Very Large Data Base Endowment*, 1(1) :761–772, 2008.

[dbm] The db-main official website. `http://www.db-main.be`.

[DT03] Alin Deutsch and Val Tannen. Mars : A system for publishing xml from mixed and redundant storage. In *In VLDB*, pages 201–212, 2003.

[Eyr08] Rémi Eyraud. Base de données - les vues et les triggers. `http://www.lif.univ-mrs.fr/~reyraud/BD/BD\_Cours\_5\_2008-2009.pdf`, 2008.

[Ger03] Olivier Gerbé. Bases de données de l'entreprise - développement d'applications : vues, contraintes d'intégrité. `http://zonecours.hec.ca/documents/A2007-1-1356112.H2006DeveloppementApplication-1.ppt`, 2003.

[Gro08] The PostgreSQL Global Development Group. *PostgreSQL 8.3.7 Documentation*, 2008.

[gRR95] Young gook Ra and Elke A. Rundensteiner. A transparent object-oriented schema change approach using view evolution. In *In IEEE International Conference on Data Engineering*, pages 165–172, 1995.

[Hai89] Jean-Luc Hainaut. A generic entity-relashionship model. 1989.

[Hai00] Jean-Luc Hainaut. *Bases de données et modèles de calcul.* Dunod, 2nd edition, 2000.

[Hai02] Jean-Luc Hainaut. *LIHD - Cours d'Ingénierie des bases de données.* 2002.

[Hai09] Jean-Luc Hainaut. *Bases de données.* Dunod, 2009.

[HH06] Jean-Marc Hick and Jean-Luc Hainaut. Database application evolution : a transformational approach. *Data and Knowledge Engineering*, 59(3) :534–558, 2006.

[HHE+99] Jean-Marc Hick, Jean-Luc Hainaut, Vincent Englebert, Didier Roland, and Jean Henrard. Stratégies pour l'évolution des applications de bases de données relationnelles : l'approche db-main. In *INFormatique des ORganisations et Systèmes d'Information et de Décision*, pages 35–54, 1999.

[Hic01] Jean-Marc Hick. *Evolution d'applications de bases de données relationnelles : Méthodes et Outils.* 2001.

[IBM09] IBM. *DB2 v9.1 Documentation*, 2009.

[Lit97] Witold Litwin. Sql - catalogues, vues, autorisations, déclencheurs. `http://ceria.dauphine.fr/cours98/CoursBD/SQL2-97.ppt`, 1997.

[Ora09] Oracle. *Oracle Database 10g Release 2 (10.2) Documentation*, 2009.

[Sch05] Robin Schumacher. Etude pratique du cache de requêtes mysql. `http://maximilian.developpez.com/mysql/queryCache/`, 2005.

[Sof09] Janus Software. *Firebird 2.0 Online Manual*, 2009.

Annexe A

Annexe - Etude de cas

Cette annexe présente, pour chacune des transformations, le détail des temps d'excécution de chacune de requêtes de l'étude de cas (Chapitre 7).

A.1 Résultats détaillés

A.1.1 Transformation T1

T1 ≡ Renommer la table CLIENT en CUSTOMER

T1(Renommer table)/R1(SELECT)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	31 ms	31 ms	31 ms
100	31 ms	30 ms	31 ms
1000	47 ms	48 ms	47 ms
10000	141 ms	138 ms	146 ms
100000	813 ms	809 ms	828 ms

T1(Renommer table)/R2(INSERT)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	31 ms	31 ms	31 ms
100	31 ms	31 ms	31 ms
1000	47 ms	56 ms	47 ms
10000	62 ms	63 ms	59 ms
100000	125 ms	125 ms	131 ms

T1(Renommer table)/R3(UPDATE)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	31 ms	31 ms	31 ms
100	31 ms	30 ms	31 ms
1000	47 ms	48 ms	43 ms
10000	141 ms	156 ms	146 ms
100000	483 ms	468 ms	469 ms

T1(Renommer table)/R4(DELETE)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	31 ms	31 ms	31 ms
100	30 ms	30 ms	31 ms
1000	47 ms	48 ms	47 ms
10000	157 ms	156 ms	146 ms
100000	453 ms	453 ms	462 ms

T1(Renommer table)/R5(ALTER)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	79 ms	erreur	62 ms
100	94 ms	erreur	79 ms
1000	141 ms	erreur	109 ms
10000	654 ms	erreur	563 ms
100000	2341 ms	erreur	2688 ms

erreur = `'renameTableCbl10.client' is not BASE TABLE`

T1(Renommer table)/R6(CREATE INDEX)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	78 ms	erreur	78 ms
100	93 ms	erreur	93 ms
1000	235 ms	erreur	203 ms
10000	803 ms	erreur	984 ms
100000	3344 ms	erreur	3296 ms

erreur = `'renameTableCbl10.client' is not BASE TABLE`

A.1.2 Transformation T2

T2 ≡ Renommer la colonne `Adresse` de `CLIENT` en `NumRue`

T2(Renommer colonne)/R1(SELECT)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	15 ms	16 ms	16 ms
100	31 ms	30 ms	31 ms
1000	47 ms	48 ms	46 ms
10000	141 ms	146 ms	146 ms
100000	735 ms	750 ms	751 ms

T2(Renommer colonne)/R2(INSERT)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	16 ms	16 ms	16 ms
100	31 ms	31 ms	31 ms
1000	30 ms	38 ms	31 ms
10000	67 ms	62 ms	63 ms
100000	121 ms	123 ms	125 ms

T2(Renommer colonne)/R3(UPDATE)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	16 ms	15 ms	15 ms
100	31 ms	30 ms	31 ms
1000	47 ms	46 ms	47 ms
10000	156 ms	156 ms	146 ms
100000	438 ms	469 ms	446 ms

T2(Renommer colonne)/R4(DELETE)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	16ms	16 ms	15 ms
100	15 ms	30 ms	15 ms
1000	47 ms	47 ms	47 ms
10000	188 ms	175 ms	168 ms
100000	468 ms	469 ms	486 ms

T2(Renommer colonne)/R5(ALTER)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	62 ms	erreur	63 ms
100	79 ms	erreur	79 ms
1000	140 ms	erreur	109 ms
10000	578 ms	erreur	578 ms
100000	2578 ms	erreur	2875 ms

erreur = `'renameColumnCbl10.client' is not BASE TABLE`

T2(Renommer colonne)/R6(CREATE INDEX)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	78 ms	erreur	78 ms
100	78 ms	erreur	78 ms
1000	204 ms	erreur	235 ms
10000	656 ms	erreur	656 ms
100000	3563 ms	erreur	3891 ms

erreur = `'renameColumnCbl10.client' is not BASE TABLE`

A.1.3 Transformation T3

T3 ≡ Scinder la colonne `Adresse` de `CLIENT` en deux colonnes `Num` et `Rue`

T3(Scinder colonne)/R1(SELECT)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	16 ms	15 ms	15 ms
100	31 ms	30 ms	31 ms
1000	47 ms	46 ms	47 ms
10000	156 ms	156 ms	146 ms
100000	438 ms	469 ms	444 ms

T3(Scinder colonne)/R2(INSERT))			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	16 ms	erreur	15 ms
100	15 ms	erreur	16 ms
1000	16 ms	erreur	16 ms
10000	78 ms	erreur	78 ms
100000	110 ms	erreur	109 ms

erreur = `The target table CLIENT of the INSERT is not insertable-into`

T3(Scinder colonne)/R3(UPDATE)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	47 ms	erreur	46 ms
100	47 ms	erreur	46 ms
1000	47 ms	erreur	46 ms
10000	110 ms	erreur	156 ms
100000	500 ms	erreur	547 ms

erreur = `Column 'Adresse' is not updatable`

T3(Scinder colonne)/R4(DELETE)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	31 ms	31 ms	31 ms
100	31 ms	31 ms	31 ms
1000	47 ms	48 ms	47 ms
10000	101 ms	93 ms	94 ms
100000	439 ms	446 ms	469 ms

T3(Scinder colonne)/R5(ALTER)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	67 ms	erreur	78 ms
100	47 ms	erreur	78 ms
1000	47 ms	erreur	94 ms
10000	172 ms	erreur	375 ms
100000	1797 ms	erreur	1844 ms

erreur = `'splitColumnCbl10.client' is not BASE TABLE`

T3(Scinder colonne)/R6(CREATE INDEX)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	63 ms	erreur	46 ms
100	63 ms	erreur	47 ms
1000	63 ms	erreur	78 ms
10000	594 ms	erreur	581 ms
100000	2406 ms	erreur	2156 ms

erreur = `'splitColumnCbl10.client' is not BASE TABLE`

A.1.4 Transformation T4

T4 ≡ Scinder la table `CLIENT` en créant une nouvelle table `ADRESSE` contenant la colonne `Adresse` de `CLIENT`

T4(Scinder table)/R1(SELECT)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	31 ms	31 ms	31 ms
100	31 ms	30 ms	31 ms
1000	47 ms	48 ms	47 ms
10000	148 ms	257 ms	242 ms
100000	701 ms	1510 ms	1438 ms

T4(Scinder table)/R2(INSERT))			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	31 ms	erreur	31 ms
100	31 ms	erreur	31 ms
1000	31 ms	erreur	31 ms
10000	80 ms	erreur	80 ms
100000	102 ms	erreur	110 ms

erreur = `Can not modify more than one base table through a join view 'splitTableCbl10.client'`

T4(Scinder table)/R3(UPDATE)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	47 ms	erreur	46 ms
100	47 ms	erreur	46 ms
1000	47 ms	erreur	63 ms
10000	109 ms	erreur	234 ms
100000	437 ms	erreur	1156 ms

erreur = `Can not modify more than one base table through a join view 'splitTableCbl10.client'`

T4(Scinder table)/R4(DELETE)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	31 ms	erreur	32 ms
100	32 ms	erreur	31 ms
1000	31 ms	erreur	32 ms
10000	125 ms	erreur	180 ms
100000	472 ms	erreur	508 ms

erreur = `Can not delete from join view 'splitTableCbl10.client'`

T4(Scinder table)/R5(ALTER)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	62 ms	erreur	62 ms
100	79 ms	erreur	79 ms
1000	78 ms	erreur	94 ms
10000	273 ms	erreur	343 ms
100000	1890 ms	erreur	1469 ms

erreur = `'splitTableCbl10.client' is not BASE TABLE`

T4(Scinder table)/R6(CREATE INDEX)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	63 ms	erreur	46 ms
100	64 ms	erreur	47 ms
1000	63 ms	erreur	78 ms
10000	594 ms	erreur	437 ms
100000	2503 ms	erreur	2681 ms

erreur = `'splitTableCbl10.client' is not BASE TABLE`

A.1.5 Transformation T5

T5 ≡ Fusionner les tables `CLIENT` et `ADRESSE` en une seule table `CLIENT` contenant toutes les colonnes des deux tables

T5(Fusionner tables)/R1(SELECT)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	62 ms	47 ms	47 ms
100	62 ms	47 ms	47 ms
1000	78 ms	47 ms	47 ms
10000	390 ms	219 ms	184 ms
100000	1503 ms	1094 ms	908 ms

T5(Fusionner tables)/R2(INSERT))			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	31 ms	erreur	31 ms
100	32 ms	erreur	31 ms
1000	31 ms	erreur	32 ms
10000	80 ms	erreur	78 ms
100000	110 ms	erreur	110 ms

erreur = `Field of view 'mergeTableCbl10.client' underlying table doesn't have a default value`

T5(Fusionner tables)/R3(UPDATE)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	31 ms	erreur	31 ms
100	31 ms	erreur	31 ms
1000	31 ms	erreur	31 ms
10000	198 ms	erreur	109 ms
100000	578 ms	erreur	515 ms

erreur = `The definition of table 'CLIENT' prevents operation UPDATE on table 'ADRESSE'`

T5(Fusionner tables)/R4(DELETE)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	31 ms	47 ms	47 ms
100	32 ms	46 ms	47 ms
1000	31 ms	46 ms	46 ms
10000	188 ms	203 ms	125 ms
100000	515 ms	472 ms	454 ms

T5(Fusionner tables)/R5(ALTER)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	68 ms	erreur	63 ms
100	78 ms	erreur	78 ms
1000	78 ms	erreur	78 ms
10000	344 ms	erreur	203 ms
100000	1469 ms	erreur	1891 ms

erreur = `'mergeTableCbl10.client' is not BASE TABLE`

T5(Fusionner tables)/R6(CREATE INDEX)			
Nb. lignes	BD Source	BD Cible	
		Avec vues	Sans vues
	Tps exec.	Tps exec.	Tps exec.
10	46 ms	erreur	62 ms
100	62 ms	erreur	78 ms
1000	156 ms	erreur	147 ms
10000	422 ms	erreur	543 ms
100000	1969 ms	erreur	2375 ms

erreur = `'mergeTableCbl10.client' is not BASE TABLE`